Ordered Model Transformations

Terrell, Jeffrey William

Awarding institution:
King's College London

University of London

Ordered Model Transformations

by

Jeffrey Terrell

July, 2013

A thesis submitted in partial fulfilment for the
degree of Doctor of Philosophy

Department of Informatics
King's College London, University of London

To ***Anne***, ***Jennifer*** and ***Michael*** with all my love.

Contents

List of Figures

Acknowledgements

I am indebted to my supervisors Iman Poernomo and Maribel Fernández: Iman for inviting me to join his research group in the first place; Maribel for stepping into the breach when Iman left King's to take up a position in industry. I am particularly grateful to Maribel for guiding me through what was a very difficult writing-up period, when with little more than five months to go, I was on the verge of giving up. I admire the calm but determined way she handled the situation.[1]

I am also grateful to Steffen Zschaler, Kevin Lano and Ondrej Rypacek for their support, particularly Ondrej, who I badgered on an almost daily basis about all matters type theoretical; Kennedy Carter and Abstract Solutions for allowing me to take time off work to pursue my dream; and David Clark and José Meseuger for examining this thesis.

Finally, I would like to thank my long-suffering wife Anne for being so unbelievably supportive during good times and bad, and for affording me the opportunity to experience something truly special in the twilight of my career. In spite of all the trials and tribulations along the way, I would not have missed it for the world.

Jeffrey Terrell
July 2014

[1] The issue at stake was fundamental: I was not prepared to write up a solution to the problem that lies at the heart of this thesis because I did not believe in its integrity. Fortunately, a superior solution presented itself—as if by divine intervention—in the nick of time.

Abstract

The rise of the model as a first class entity in the field of software engineering, sparked an intense period of research into the design and development of model transformations in the 1990s and 2000s. This thesis is about a particular kind of model transformation, in which the source and target models are ordered by containment. It asserts that the monolithic proof of correctness of an ordered set of transformations is equivalent to the sum of the proofs of its elemental parts. It also demonstrates by means of a type theoretical solution that the assertion is sound.

1 Introduction

This thesis is about a particular kind of model transformation, in which the source and target models are ordered by containment. It asserts that the monolithic proof of correctness of an ordered set of transformations is equivalent to the sum of the proofs of its elemental parts. This introductory chapter outlines the factors which motivated the author's interest in this line of research, gives an overview of the problem it sought to address and the solution that eventually emerged, states the contributions which the author claims to have made in the field of study, and provides a summary of the chapters which follow.

1.1 Motivation

By way of introduction, compare and contrast the traditional elaborative approach to software development with the more recent translational approach, as shown in Figure 1.1.

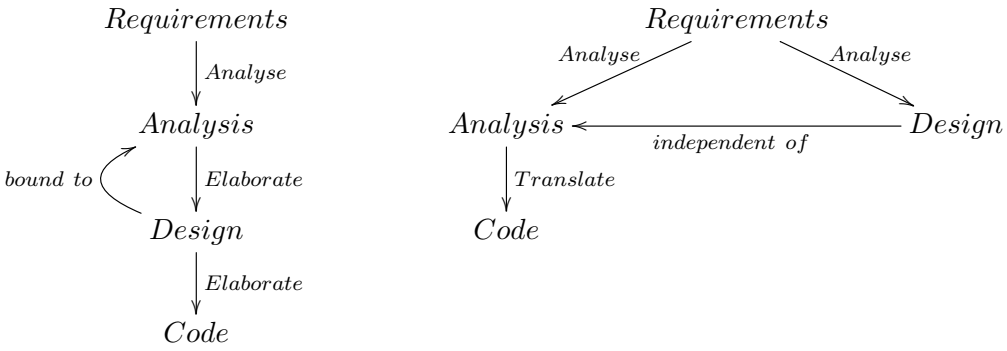

Figure 1.1: Elaborative (left) and translational (right) approaches to software development.

In the *elaborative* approach, an analysis model is manually elaborated into a design model by incrementally adding design information, such as details of tasking, messaging, data structures, data distribution and so on. Although the process may require several iterations, once the design model is complete, it can be manually elaborated into code by adding implementation detail. There are at least two disadvantages with this approach. First, the cost of manually maintaining a fact in three places (analysis, design and code) is prohibitively expensive. As a result, most organisations do not insist on rippling changes in requirements down the life cycle through the analysis and design models and into the

code, opting instead simply to update the code, particularly if deadlines are looming. Subsequently, once the first code is released, the code itself becomes the primary specification of a system's behaviour, and the analysis and design models gradually fall into obsolescence. Second, the system will be harder to maintain. This is particularly so if there are no design strategies in place, because each component of the analysis model may be elaborated in a different way.

In the *translational* approach, "design" is considered to be a subject matter for analysis in its own right, just like any other. But unlike the elaborative approach, it is not inextricably bound to the elaboration of any particular analysis model. At worst, it takes the form of a set of rules and policies that prescribe how the components of an analysis model should be manually translated into code. At best, it is a tool[1] that *automatically* translates the components of an analysis model into code. Such tools are enormous corporate assets, because like compilers they are inherently reusable.

Since 2001, the author has been involved in developing industrial-strength tools—of the kind described above—for the Model Driven Architecture (MDA) [61] market. Each tool transforms an executable subset of the Unified Modelling Language (UML) [62] to a particular target platform. Given the relative immaturity of this technology, it is surprising that not once has the integrity of these tools been called into question, even by clients in high profile industries such as telecommunications, defence and space.

To the obvious advantages of a translational approach over an elaborative approach, one must add a severe warning. What happens if there is a bug in the translator? Of course, this question could be asked of any software product. However, it is a question that is particularly pertinent to ask of translators, because if a translation rule has not been prototyped correctly, or there is a nuanced coding error in the translator, a bug—which could lie dormant for a long period of time before it chose to show itself—could be stamped out tens, hundreds or even thousands of times during the translation of a large model.

On the basis of these considerations, the author was delighted to be offered the opportunity to conduct research in this field, as a means of adding rigour to the development of model translators (i.e. transformations), not only of the model to text variety as discussed above, but also of the model to model variety.

1.2 Problem

A significant amount of research has already been conducted in the field of model transformations, from graph-theoretical solutions (e.g. Agrawal [10]), through language design (e.g. Jouault [72]) to transformations by example (e.g. Varró [135]). However, if there is a need to give a categorical assurance that the implementation of a transformation meets its specification—which indeed there is in the field of high integrity systems—none of them comes up to expectations like a type theoretical solution. A type theoretical solution of a model transformation is inherently trustworthy because it not only contains (i) a logical specification of the transformation, and (ii) a program that implements the transformation,

[1] The tool is called a code generator in industry, and a model to text transformation in academia.

but it also contains (iii) a proof that the program satisfies the specification of the transformation. There is a price to pay for this rigour, however, in that the expertise and effort required to develop even the most modest of type theoretical solutions is considerable.

In the author's opinion, hierarchy is the natural medium for organising one's thoughts about the complexities of the real world, and many abstractions of real or hypothetical software systems are, unsurprisingly, hierarchical too. That is not to say that they are entirely hierarchical. However, it is undoubtedly the case that many systems do have a strong hierarchical basis. For these reasons, together with the author's own experience of implementing translators by stepping down the containment and generalisation hierarchies of source models, the primary purpose of this thesis is to find a way of simplifying the certification, i.e. proof of correctness, of a potentially large transformation between two hierarchical models. In virtue of its regular structure, it seems plausible to assert that the proof of correctness of such a transformation is susceptible to the principle of divide and conquer, whereby the proof of the whole is equivalent—in some sense—to the sum of the proofs of its elemental parts.

1.3 Solution

The solution which emerged from this research is obviously beyond the scope of this introductory chapter, but a flavour for what it is can be described by analogy. A ladder is composed of rungs and risers. When the ladder is upright, the rungs are horizontal and the risers are vertical. Imagine an upright ladder (see Figure 1.2) as a nested set of transformations between the classes X_i of a source model on the left, and the classes Y_i of a target model on the right, in which each rung represents a transformation T_i between X_i and Y_i, and each pair of risers R_i and S_i represents a containment relation between transformations T_i and T_{i+1}. Thus, the specification of T_1 contains the specification of T_2, which contains the specification of T_3, and so on, i.e. $T_1 \supset T_2 \supset \cdots \supset T_n$.

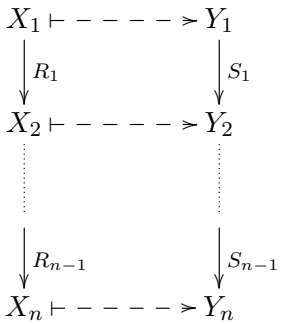

Figure 1.2: A ladder of transformations.

Let T be the type of ladder, let D_i be the type of rung i, and let L_i be the type of a pair of risers R_i and S_i. If t is a proof (i.e. a categorical assurance) that f implements the whole of T correctly, d_i is a proof that f_i implements rung D_i correctly, and l_i is a proof that f_i and f_{i+1} implement risers L_i correctly, then what emerged from this research is that t is

the composition of d_i for $i = 1, \ldots, n$, and l_i for $i = 1, \ldots, n-1$. In other words, the proof of the whole ladder is equivalent to the sum of the proofs of its rungs and risers. This is a useful result in that the latter (the parts) are easier to derive than the former (the whole). This result extends to trees of ladders as well.

1.4 Contributions

The author claims to have made the following contributions to the field of model transformations.

1. Devised a method for proving the correctness of an arbitrarily large ordered model transformation, by showing how its proof can be assured by summing the proofs of its parts, thereby making it easier to derive.

2. Raised the prospect of decomposing a class of transformations into a set of horizontal and vertical components, where each horizontal component is responsible for transforming data between the source and target models, and each vertical component is responsible for transforming links.

3. Formulated a means of defining recursive specifications of model transformations in type theory.

4. Composed a type system for implementing model transformations.

The author has also published papers with Fernández [54], Poernomo [113] and Zschaler [143] on earlier ideas related to this thesis.

1.5 Method

A set of graded example transformations were devised with which to explore the issues. Initially, they were coded in a mathematical reasoning tool called Coq [16], to prove their correctness. However, the author was minded from the outset to develop the transformations in a neutral language, so that they could be ported to other tools if necessary, and consequently converted them by hand to a variant of Martin-Löf [85] type theory. During the exploration of one particular case study—a kind of hierarchical transformation—the author noticed how certain segments of the proof repeated themselves over and over, as the proof descended the containment hierarchy. This led to the realisation that it may be possible to factorise the proof in some way.

Apart from Poernomo [110, 112], who pioneered the work in this field, the author is aware of only one other group [31, 30] which has conducted research into the use of constructive type theory to implement model transformations. Of course, this could be regarded as an opportunity rather than a hinderance.

1.6 Organisation

This thesis contains 8 chapters. *Chapter 2* is devoted to type theory, which is essentially a λ-calculus with dependent types. It starts with a historical perspective, is followed by an overview of the typed and untyped λ-calculi, and concludes with a rather metronomic description of a type system for model transformations, which the reader is advised to skim on first reading. Type theory appears to be unique amongst model transformation languages in that it not only provides the means to specify and implement model transformations, but it also provides the means to prove them as well. What is impressive is that this all takes place within the same formalism.

Chapter 3 introduces the reader to software models via the standard documents in the field, i.e. those of the Object Management Group (OMG). It also contains a detailed formalisation of a particular kind of model—the class model— in type theory.

Chapter 4 describes the encoding of a number of model transformations in type theory, including one with a recursive specification.

Chapters 5 and 6 are analogues of each other. The former details the construction of an (uncertified) ordered model transformation from a set of dependent data types, and the latter does likewise for a certified ordered model transformation, where a certified ordered model transformation is the composition of an (uncertified) ordered model transformation and a set of proof terms, which when extracted are sufficient to prove the correctness of the (uncertified) ordered model transformation. The basic idea is that the construction of a certified ordered model transformation is tantamount to proving the correctness of the corresponding (uncertified) ordered model transformation. The logical interpretation theorem in Chapter 6, which is a formalisation of this idea, is the author's main contribution.

Finally, *Chapter 7* overviews the related work in the field, and *Chapter 8* sums up and suggests areas for further study.

2 Type Theory

The primary purpose of this thesis is to find a type theoretical way of simplifying the certification of a particular kind of model transformation, in which the source and target models exhibit a strong sense of hierarchy. This chapter provides an introduction to the underlying theory. It starts by discussing the rather simple yet powerful principle on which type theory is based—that every object has a type—and goes on to describe how and why type theory originated. It continues with an overview of the lambda calculus [36, 67], the programming language [84] on which modern type theory is based. However, the bulk of this chapter is taken up in defining a system of types for certifying model transformations, along the lines of Thompson's system TT [133].

2.1 Basic Concepts

Informally, a type is a collection of objects, and conversely, an object is an *inhabitant* of a type. An object never exists on its own account, only in virtue of its relation with a certain type [85]. A type is formally defined by prescribing how its *canonical* objects are constructed, and what it means for two objects to be equal [95, 17].

- *(Canonical Objects)* A canonical object of a type is one that *cannot* be reduced to a simpler form. For example, the canonical objects of Nat—the type of natural numbers—are constructed by means of the following rules.
  - 0 is an object of Nat.
  - The successor of n, i.e. $succ(n)$, is an object of Nat, if n is an object of Nat.

  Applying the first rule, and then repeatedly applying the second rule, gives the canonical objects of Nat, i.e.

  $$0,\ succ(0),\ succ(succ(0)),\ succ(succ(succ(0))),\ \ldots\ ,$$

  or $0, 1, 2, \ldots$ as they are usually (and henceforth will be) denoted. An alternative encoding of the natural numbers, as a sequence of lambda terms, is provided by Church [38].
- *(Equal Objects)* A non-canonical object of a type is one that *can* be reduced to a simpler form, that is to say the form of a canonical object. For example, the expression

$1 + 2$, a non-canonical object of type Nat, computes to 3, a canonical object of type Nat. Since $1 + 2$ and 3 are computationally equal, this suggests a possible definition for what it means for two objects to be equal.

Definition 1. *(Equal Objects)* Two objects are equal if they reduce to the same canonical object.

The equality relation on a type divides the totality of objects of that type into a set of equivalence classes, in which each class contains the objects that compute to a particular canonical object. For example, the non-canonical objects $1 + 2$ and 1×3 are in the same equivalence class as the canonical object 3, as indeed are many others.

Nat is an example of a type with an infinite number of canonical objects. However, not all types have that many. $Bool$, for example, has only two ($False$ and $True$); $\top$ has only one ($Triv$); and $\bot$ has none at all. (The logical interpretations of $\top$ and $\bot$, namely truth and falsity, should not be confused with the objects of $Bool$.)

A type system is *open* in the sense that a new type can be introduced as and when required, either by defining rules like Nat, or by combining existing types with logical connectives. For example, $Bool \to Nat$ is the type of functions that take an object of $Bool$ to an object of Nat. One function that satisfies this rather weak specification is

$$\lambda b\colon Bool\,.\,if\ b\ then\ 1\ else\ 2\ ,$$

but clearly there are many others. The lambda calculus, of which this function is a term, is discussed in Section 2.5.

The notion of a type is sufficiently broad as to encompass not only types whose objects are atomic, like 3 is to Nat, but also types whose objects are themselves types, like Nat is to its type U_0, the universe of small types, and like U_0 is to its type U_1, the first universe of large types, and so on. The hierarchical structure of universes is discussed in Section 2.6.1.

The relation "is an object (or inhabitant) of" is denoted by the colon symbol, and so $a\colon A$ is an assertion that a is an object of A. What this means in practice may vary from one context to another. For example, in the context of propositional logic, $a\colon A$ may be interpreted as "a is a proof of proposition A", whereas in the context of certified program development, $a\colon A$ may be regarded as "a is a program that meets specification A". It is hardly surprising that different interpretations of $a\colon A$ exist, because the notion of an object in relation to a type, can be explained in much the same way as a proof in relation to a proposition, and a program in relation to a specification, since a proposition is provable if its representation as a type is inhabited, and a specification is satisfiable if its representation as a type is inhabited. Having established these similarities, it is worth noting a difference. In proving a proposition, the requirement is simply to find an object (proof); any object will do, so long as one exists. However, in proving that a program satisfies a specification, the requirement may not only be to find an object (program), but also to find an *efficient* object; and if there are many objects to choose from, one object will inevitably be preferred to the others.

2.2 Origins

The first type theory to appear in print appears to have been Russell's *Doctrine of Types* [122]. In Russell's own words, it was "put forward tentatively, as affording a possible solution to the contradiction", the contradiction being what later became known as Russell's Paradox. Stated briefly and paraphrasing Russell, "it is meaningless to assert something about *all* cases of some kind, if from what is said a new case is generated, which both is and is not of the same kind as the cases of which *all* were concerned". The most celebrated example of Russell's Paradox is the set of all sets w that do not contain themselves as members, i.e.

$$w \equiv_{df} \{x \colon x \notin x\}.$$

This fits Russell's description of a statement about all cases of some kind, the kind being sets. Furthermore, the new case that "seems to be generated" is w, which Russell asserts "both is and is not of the same kind" as the set of all sets that do not contain themselves as members. In other words, w both is and is not a member of itself. If w is assumed to be a member of itself, it follows that w is not a member of itself, and vice versa: clearly a contradiction.

Russell eventually formulated a solution to the contradiction by introducing a hierarchy of types, where a type was defined as the collection of values for which a function has values, the principle being that "whatever contains an apparent variable must be of a different type from the possible values of that variable", that is to say it must be of a higher type. As Monk describes in his excellent biography [91], Russell's preoccupation with the contradiction was an all consuming affair, which led to a long period of introspection and self doubt, and hampered his attempts to reconstruct the foundations of mathematics.

In establishing the rules of type formation, it is important to consider whether or not it makes sense for a type to be an object of itself. Is it paradoxical? Russell clearly thought it was. However, almost seventy years later, Martin-Löf, the acknowledged founder of modern type theory, proposed a theory of types for the formalisation of intuitionistic mathematics (later revised in [85] and [83]), based on an "axiom of all types whatsoever, which is at the same time a type and an object of that type", only to discover the following year, courtesy of a young French mathematician Girard [58], that his theory was inconsistent.

Russell's *Doctrine of Types* evolved into the *Ramified Theory of Types* [123], and then thanks to Church, into *The Simple Theory of Types* [38]. However, it was not until much later that modern type theory began to emerge. In his 1958 book on *Combinatorial Logic* [44], Curry observed "a striking analogy between the theory of functionality and the theory of implication in propositional algebra". In the theory of functionality, the basic combinators K and S are given by

$$\begin{aligned} K &=_{df} \lambda x \,.\, \lambda y \,.\, x \\ S &=_{df} \lambda x \,.\, \lambda y \,.\, \lambda z \,.\, x\ z\ (y\ z)\ . \end{aligned}$$

If α is the category of x, and β is the category of y (Curry called α and β categories, not types), then clearly K is a function that takes α to a function that takes β to α, i.e.

$$K \colon \alpha \to (\beta \to \alpha)\ .$$

Less obvious, perhaps, is that

$$S\colon (\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma)) \ .$$

Curry observed that if the symbol $\to$ is interpreted not as function application but as logical implication, then K and S are the axioms of the implicational propositional calculus, namely

$$\begin{array}{l} \alpha \to (\beta \to \alpha) \\ (\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma)) \ . \end{array}$$

Howard later extended Curry's ideas in *The Formulae-as-Types Notion of Construction* [68], in which he defined the *construction* of a sequent of the form $\Gamma \vdash \alpha$ as a term of type α. He also defined two rules of term formation, *abstraction* and *application*, which mirror the rules of inference in the implicational propositional logic, i.e.

$$\begin{array}{c} \dfrac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta} \\[2ex] \dfrac{\Gamma \vdash \alpha \quad \Delta \vdash \alpha \to \beta}{\Gamma, \Delta \vdash \beta} \end{array} \ .$$

From the first rule he defined abstraction:[1]

$$\text{from } y^{\beta}, \text{ get } (\lambda x^{\alpha}.y^{\beta})^{\alpha \to \beta} \ ;$$

and from the second rule he defined application:

$$\text{from } f^{\alpha \to \beta} \text{ and } x^{\alpha}, \text{ get } (f^{\alpha \to \beta}\ x^{\alpha})^{\beta}.$$

With regards the other logical connectives ($\wedge$, $\vee$ and $\neg$), Howard added so-called *prime* terms to the base set, and then proceeded to invoke the application rule. For example, with regards $\wedge$, he added

$$f_1^{\,\alpha \wedge \beta \to \alpha}, \ f_2^{\,\alpha \wedge \beta \to \beta} \text{ and } f_3^{\,\alpha \to (\beta \to \alpha \wedge \beta)} \ ;$$

then he invoked the application rule three times, i.e.

$$\text{from } p^{\alpha \wedge \beta}, \text{ get } (f_1^{\,\alpha \wedge \beta \to \alpha}\ p^{\alpha \wedge \beta})^{\,\alpha} \ ;$$

$$\text{from } p^{\alpha \wedge \beta}, \text{ get } (f_2^{\,\alpha \wedge \beta \to \beta}\ p^{\alpha \wedge \beta})^{\,\beta} \ ;$$

$$\text{from } p^{\alpha} \text{ and } q^{\beta}, \text{ get } (f_3^{\,\alpha \to (\beta \to \alpha \wedge \beta)}\ p^{\alpha}\ q^{\beta})^{\,\alpha \wedge \beta} \ .$$

The correspondence between propositions and types, which Curry first observed and Howard subsequently extended, is known as the Curry-Howard Isomorphism.

[1] Howard used superscripts to denotes types.

2.3 Constructive Mathematics

In 1967, Bishop wrote a book [17] on the foundations of constructive analysis, which one reviewer—Myhill [94]—described as "the most important book on constructive mathematics ever written". In it, Bishop put forward a manifesto for the constructivisation of mathematics, "to hasten the inevitable day when constructive mathematics will be the accepted norm". The issue at stake—the meaning of certain proofs on which much of classical mathematics is based, whereby as Beeson [15] says, "one proves that something exists by assuming that it does not exist, and then derives a contradiction without showing a way to construct the thing in question"—is one that has been debated by the mathematical community for over 150 years.

To understand the real significance of Bishop's book, one needs to understand the history of constructivism, and the summary by Robinson [120] provides a useful starting point. "In the second half of the nineteenth century, Leopold Kronecker made a determined effort to turn mathematics away from its trend of ever increasing abstraction. His approach was based on the principle that in order to be meaningful, an existential assertion has to be buttressed by the actual construction of the object in question. Thus, a procedure that leads us to infer the existence of a mathematical object from purely formal-deductive considerations, e.g. by the use of the principle of the excluded middle, is regarded as inadequate or even misleading." What Robinson failed to point out—so says Beeson—was that "everyone else [apart from Bishop] believed that if one accepted a constructivist philosophy, then one would have to give up much of classical mathematics". Bishop showed that this was not the case.

The most sustained attempt to constructivise mathematics after Kronecker was made by Brouwer [26], beginning in 1907. However, as Bishop notes, the movement Brouwer founded failed to convince the mathematical community that "abandonment of the idealistic viewpoint would not steralize or cripple the development of mathematics", and that Brouwer was "much more successful in his criticism of classical mathematics than in his efforts to replace it with something better".

The distinction between constructive and non-constructive proofs (to paraphrase Dummett [49]) arises out of proofs of disjunctive and existential statements. A proof of the disjunction $A \vee B$ is said to be constructive if it is a proof of A or a proof of B, or an effective means of constructing a proof of one or other disjunct; and a proof of the existential $\exists a\colon A\,.\,P(a)$ is said to be constructive if it is not only a proof that P holds for some a, but also a declaration as to what that a is, or an effective means of constructing it. [2]

The proofs of correctness of the model transformations in this thesis are all constructive.

[2] Consider the proposition, "there are irrational numbers a and b such that a^b is rational". Now, $\sqrt{2}$ is irrational and $\sqrt{2}^{\sqrt{2}}$ is either rational or irrational. If one assumes that $\sqrt{2}^{\sqrt{2}}$ is rational, and let $a = b = \sqrt{2}$, then, by assumption, the proposition is proved. On the other hand, if one lets $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$, then $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2$, which is rational: again, the proposition is proved. The proposition has therefore been proved on the basis that ($\sqrt{2}^{\sqrt{2}}$ is rational $\vee$ $\sqrt{2}^{\sqrt{2}}$ is irrational), without knowing whether $\sqrt{2}^{\sqrt{2}}$ is rational or not. The basis of this proof, which is an example of the use of the law of the excluded middle, is not valid in a constructive setting.

Each one—notionally at least—comes with a certificate (i.e. a proof) which shows how the program that implements it meets its specification.

2.4 Coq Proof Assistant

This section provides a brief overview of the tool that was used to validate the proofs of correctness in this thesis. The Coq proof assistant [16] is a rich development environment for formally proving mathematical assertions, which has been widely adopted by the research community, and is rapidly becoming the tool of choice for teaching the foundations of mathematics and computer science. Coq—as it is usually called—is well suited to the needs of developing programs in which absolute trust is required. In fields as diverse as transportation, telecommunications, energy, banking and so on, the need for programs to conform rigorously to inviolable specifications justifies the effort required to formally verify them.

Coq implements a program specification language called *Gallina*, which is based on the Calculus of Inductive Constructions [42, 106], the synthesis of a richly-typed functional programming language and a higher-order logic, i.e. a form of constructive type theory. Through a *Vernacular* language of commands [132], Coq allows users to

- define functions and predicates;
- assert the truth of mathematical theorems and program specifications;
- interactively develop formal proofs of theorems and specifications;
- extract efficient programs in languages such as Objective Caml and Haskell.

The Coq proof of an assertion such as $\vdash \forall x\colon Nat\,.\,0 \leq x$, is composed of a sequence of *tactics*, where each tactic employs backward reasoning to break down the assertion into simpler and simpler parts, until such parts are obtained that can be trivially proved. For example:

Coq Listing 1. *(Simple Proof)* The assertion `0 <= 2` breaks down into `0 <= 1` and then `0 <= 0` by applying the object `le_S`. The assertion `0 <= 0` is then proved by applying the `le_n` object. The objects `le_S` and `le_n` are the constructors of type `le`, Coq's less than or equal type.

```
Goal 0 <= 2.
Proof.
   apply le_S.
   (new goal 0 <= 1)
   apply le_S.
   (new goal 0 <= 0)
   apply le_n.
Qed.
```

Listings such as the one above, are frequently used to provide evidence that the material in this thesis is sound, and to equip readers with an alternative view of the subject matter, which may aid understanding.

In 2013, the Coq development team won an ACM SIGPLAN software award in recognition of the role that Coq has played—and is continuing to play—in the era of "formal assurance in mathematics, semantics and program verification".

2.5 λ-Calculus

The λ-calculus is a small yet powerful functional programming language. It was invented by Church in the 1930s as part of a study into the general properties of functions, which he later developed into a set of postulates for the foundation of logic [37]. Unfortunately for him, the theory he proposed had to be abandoned a few years later when two of his former students Kleene and Rosser [76] found it to be inconsistent.[3] However, the function notation he invented—which Church claimed was explicitly motivated by the study—lives on in many different forms. Broadly speaking, it falls into two main categories: untyped and typed.

2.5.1 Untyped λ-Calculus

The symbols of the untyped λ-calculus are drawn from

$$\lambda \quad . \quad (\quad)$$

and a countably infinite set of variables, ranged over by meta-mathematical variables $x, y, z, \ldots$.[4] The distinguished symbol sequences (λ-terms) are defined as follows.

Definition 2. *(λ-terms)*

1. *(Variable)* All variables are λ-terms.
2. *(Application)* If M and N are λ-terms, then $(M\ N)$ is a λ-term.
3. *(Abstraction)* If M is a λ-term, and x is a variable, then $(\lambda x\,.\,M)$ is a λ-term.

Example 1. *(λ-terms)* If x and y are variables, then by the variable rule, x and y are λ-terms; and by the application rule, $(x\,y)$ and $((x\,y)\,y)$ are λ-terms; and by the abstraction rule, $(\lambda x\,.\,x)$ and $(\lambda x\,.\,(\lambda y\,.\,(x\,y)))$ are λ-terms.

[3] A theory is inconsistent if a proposition and its negation can both be shown to hold. A consistent theory is one without a contradiction.

[4] There are no hard and fast rules as to how the variables should be constructed. Hindley and Seldin [67], for instance, admit the symbols v and $_0$, and define them to be

$$\{v_0, v_{00}, v_{000}, \ldots\}\ ,$$

while Church [36] admits the symbols $a, \ldots z$, and the over-line symbol, and defines them to be

$$\{a, \ldots, z, \overline{a}, \ldots, \overline{z}, \overline{\overline{a}}, \ldots \overline{\overline{z}}, \ldots\}\ .$$

The abundance of parentheses in λ-terms makes them unwieldy and difficult to read. To alleviate this problem, the following syntactic conventions—which are mostly due to Thompson [133]—have been adopted. First, the outermost parentheses are considered to be redundant, so that $x\,y$ means $(x\,y)$ and $\lambda x\,.\,y$ means $(\lambda x\,.\,y)$. Second, application binds more tightly than abstraction, so that $\lambda x\,.\,x\,y$ means $\lambda x\,.\,(x\,y)$ and not $(\lambda x\,.\,x)\,y$. Third, application associates to the left, so that $x\,y\,z$ means $(x\,y)\,z$ and not $x\,(y\,z)$. Finally, abstraction associates to the right, so that $\lambda x\,.\,\lambda y\,.\,x$ means $\lambda x\,.\,(\lambda y\,.\,x)$.

Definition 3. *(Variables)* An occurrence of a variable in a λ-term is said to be *free* if it is not in the scope of an abstraction, and *bound* otherwise. For example:

Coq Listing 2. *(Variables)* The only occurrence of x in x is free (see `P` below); the second occurence of x in $\lambda x\,.\,x$ is bound (see `Q`); and the second and third occurences of x in $(\lambda x\,.\,x)\,x$ are bound and free respectively (see `R`). Ignore the fact that `x` is typed.

```
Variable x : nat.
Definition P := x.
Definition Q := fun (x : nat) => x.
Definition R := (fun (x : nat) => x) x.
```

Definition 4. *(α-conversion)* Consider the term $\lambda x\,.\,M$, which may either stand on its own or as part of a larger term. If y is not a free variable in M, the expression formed by changing the bound variable in $\lambda x\,.\,M$ from x to y, and replacing all occurences of x in M by y, is known as an α-conversion of $\lambda x\,.\,M$, i.e.

$$\lambda x\,.\,M \to_\alpha \lambda y\,.\,M\,[y/x]\,.$$

Any rigorous treatment of the λ-calculus must include α-conversion. One common approach is to treat λ-calculus not as a calculus of λ-terms, but as a calculus of equivalence classes of λ-terms under the relation $\to_\alpha$. In that way, individual terms are simply representatives of their classes.

Definition 5. *(Semantics)* The semantics of the untyped λ-calculus are as follows.

- $\lambda x\,.\,e$ is an *abstraction* of e, i.e. a function that takes an input x to an output e, where e may or may not contain occurrences of x. For example, $\lambda x\,.\,x$ is a function that takes x to itself (the identity function), and $\lambda x\,.\,\lambda y\,.\,x$ is a function that takes x to a function that takes y to x (the K combinator).
- $(\lambda x\,.\,e)\,f$ is an *application* of $\lambda x\,.\,e$, i.e. an application of an abstraction of e to an input f, which is evaluated by substituting f for the free occurrences of x in e. For example, the value of $(\lambda x\,.\,x)\,(\lambda x\,.\,x)$ is $\lambda x\,.\,x$ (the input passed to $\lambda x\,.\,x$ is $\lambda x\,.\,x$).

Definition 6. *(Substitution)* The result of substituting f for the free occurrences of x in e, changing bound variables in e to avoid clashes with the free variables of f if necessary, is denoted by $e\,[f/x]$.

Definition 7. *(β-reduction)* The β-reduction of an application is given by

$$(\lambda x \,.\, e)\, f \to_\beta e\,[f/x]\ .$$

Coq Listing 3. *(`Eval`)* Coq supports a number of reduction strategies, including β-reduction and δ-reduction (unfolding of definitions), which can be applied individually as in line 1, or altogether as in line 3.

```
1> Eval cbv beta in (fun (x : nat) => x) 1.
     = 1
     : nat
2> Definition P := fun (x : nat) => x.
3> Compute P 1.
     = 1
     : nat
```

Definition 8. *(Reduction)* If there exists a sequence of zero or more β-reductions between e and f, i.e.

$$e \to_\beta \cdots \to_\beta f\ ,$$

then e is said to reduce to f, and the sequence is denoted by $e \twoheadrightarrow f$. Contrariwise, $f \twoheadleftarrow g$ denotes a sequence of expansions between f and g.

Definition 9. *(Conversion)* If there exists a sequence of zero or more reductions and/or expansions between e and f, then e and f are said to be convertible, and the sequence is denoted by $e \twoheadleftarrow\!\!\!\twoheadrightarrow f$. For example, $1 + 2 \twoheadleftarrow\!\!\!\twoheadrightarrow 1 \times 3$ since $1 + 2 \twoheadrightarrow 3 \twoheadleftarrow 1 \times 3$.

2.5.2 Typed λ-Calculus

The untyped λ-calculus is undoubtedly a powerful language. However, its power comes at a price in that not all λ-terms terminate. For example, if

$$\Omega \ =_{df}\ (\lambda x \,.\, x\ x)\ (\lambda x \,.\, x\ x)\ ,$$

then

$$\Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \ldots\ .$$

In contrast, the typed λ-calculus (on which modern type theory is based) does not suffer from the same problem in that all typed λ-terms terminate (by the strong normalisation theorem). To define the typed λ-terms, it is first necessary to define the notion of a simple type.

Definition 10. *(Simple Type)* Assume the existence of a set of base types like Nat and $Bool$. A simple type is either a base type, or a function type denoted by $\sigma \to \tau$, where σ and τ are simple types. A function type is intended to denote a class of functions that take an input of type σ to an output of type τ.

Definition 11. *(Typed λ-terms)* The typed λ-terms are of three kinds.

1. *(Variables)* Assume the existence of an infinite set of untyped variables as for the untyped lambda calculus. A *typed variable* is made by attaching a type as a superscript to an untyped variable, so that each typed variable has only one type, and each type is attached to every untyped variable. A variable x of type τ is denoted by x^τ.

2. *(Application)* If $M^{\sigma\to\tau}$ and N^σ are typed λ-terms, then $(M^{\sigma\to\tau}\ N^\sigma)^\tau$ is a typed λ-term.

3. *(Abstraction)* If x^σ is a typed variable and M^τ is a typed λ-term, then $(\lambda x^\sigma\,.\,M^\tau)^{\sigma\to\tau}$ is a typed λ-term.

Example 2. *(Typed λ-terms)* The identity function is given by

$$(\lambda x^\sigma\,.\,x^\sigma)^{\sigma\to\sigma}$$

for some σ. Further, the K combinator is given by

$$(\lambda x^\sigma\,.\,(\lambda y^\tau\,.\,x^\sigma)^{\tau\to\sigma})^{\sigma\to(\tau\to\sigma)}$$

for some σ and τ. These examples come from Hindley and Seldin [67].

Substitution, reduction and conversion are defined in the same way as for the untyped λ-calculus.

2.6 Type System

This section defines a suitable collection of types for proving the correctness of the model transformations in this thesis. It is founded on a hierarchy of type universes, and comprises base types, logical types, quantified types, equality types and truth and falsity types, more or less what Thompson [133] calls system TT. Each type is defined by a set of formation, introduction, elimination and computation rules.

- A formation rule defines the means of asserting "such and such an expression is a type". The expression may be as simple as the symbol A, say, in which case the formation rule asserts "A is a type", or it may be something more complex like $A \wedge B$, in which case the formation rule asserts "$A \wedge B$ is a type, if A and B are types". The expression may also contain a free variable a of type A, say, as in $C\,a$, in which case the formation rules asserts "for each object a of type A, $C\,a$ is a type", or in other words "$C\,a$ is a family of types on A". $C\,a$ is an example of a *dependent* type. The sum total of all formation rules asserts the existence of every type in the system.

- An introduction rule defines the means of constructing a subset of the canonical objects of a type. A canonical object of a type may be introduced categorically as in "0 is a canonical object of type Nat", or hypothetically as in "if n is an object of type Nat, then $succ(n)$ is a canonical object of type Nat". Note that n need not be a canonical object for $succ(n)$ to be a canonical object.

- In its simplest form, an elimination rule defines the means of destructing a type into one or other of its constituent parts, thereby removing one of the parts from a logical train of thought. For example, type $A \wedge B$ can be destructed into types A and B using the $\wedge$ elimination rule. In general, though, an elimination rule defines the means of constructing objects of a dependent type $P\,a$, say, where P is a property on an object a of type A. As part of its formulation, an elimination rule effectively enumerates the possible ways in which an object of a type can be constructed, providing a means of closure for the introduction rules.
- A computation rule defines the means of reducing a non-canonical object to a simpler form. A canonical object cannot be simplified.

2.6.1 Universes

The type system is initialised with a hierarchy of universes U_n for $n = 0, 1, 2, \ldots$, prior to the formation of any user-defined types.[5] In theory, a universe has the capacity to house any number of types, and is limited only by the user's imagination. A universe U plays two roles: (i) the role of a *type* when the focus is on what objects U houses, and (ii) the role of an *object* when the focus is on what universe houses U.

Formation Rule (U)

The formation rule of U asserts that U_n is an object of U_{n+1} for all natural numbers n, i.e.

$$\frac{}{U_n : U_{n+1}}\ (U\,F)\ .$$

Thus, "U_0 the object" is an inhabitant of "U_1 the type", "U_1 the object" is an inhabitant of "U_2 the type", and so on, as shown in Figure 2.1. The first universe U_0 is known as the universe of *small types* because its inhabitants (small types like Nat, say) house primitive objects, not types. The second and subsequent universes $U_1, U_2, \ldots$ are known as the universes of large types because their inhabitants house types. U_{n+1} houses larger types than U_n for all n.

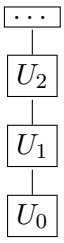

Figure 2.1: The initial state of the type system: U_0 is both an empty universe and an object of U_1; U_1 is both a one-type universe and an object of U_2, and so on.

[5] In practice, it seems likely that universes are created on demand, as and when they are required.

Coq Listing 4. *(`Set`, `Prop` and `Type`)* Luo [82] makes out the case for drawing a distinction between logical propositions and types, on the grounds that "logic has a trait of being universal" and therefore application *independent*, whereas types have a trait of being particular and therefore application *dependent*. This is the view taken by Coq, where the universe of small data types is denoted by `Set`, and the universe of propositions is denoted by `Prop`. Thereafter, `Set` and `Prop` inhabit the `Type` universe, and `Type` inhabits what appears to be itself, but which is actually a higher form of `Type` universe. The `Check` command can be used to confirm the types of the universes as follows.

```
> Check Set.
Set
     : Type
> Check Prop.
Prop
     : Type
> Check Type.
Type
     : Type
> Set Printing Universes.
> Check Set.
Set
     : Type (* (Set)+1 *)
> Check Type (* (Set)+1 *).
Type (* Top.1 *)
     : Type (* (Top.1)+1 *)
> Check Type (* (Top.1)+1 *).
Type (* Top.2 *)
     : Type (* (Top.2)+1 *)
```

2.6.2 Contexts

Most programming languages, including type theory, distinguish between the assertion "x is an object of type A" and the assertion "x is an object of type A and t is its value". For example, the statement `extern int x` in the C programming language *declares* a variable `x` of type `int`, whereas the statement `int x = 1` *defines* the value of a variable `x` of type `int` to be `1`.

Definition 12. *(Declaration)* A declaration $x : A$ is an assertion that x is an object of type A.

Definition 13. *(Definition)* A definition $x =_{df} t : A$ is an assertion that the value of an object x of type A is t.

Most programming languages also distinguish between variables that are accessible everywhere, and those that are only accessible within certain regions.

Definition 14. *(Scope)* The scope of an object is *global* if its access is unrestricted, and *local* if its access is restricted to a particular region.

Definition 15. *(Environment)* An environment is a collection of global declarations and definitions.

Definition 16. *(Context)* A context is a collection of local declarations and definitions.

Within an environment $\mathcal{E}$ and context Γ, the totality of declarations and definitions in scope is denoted by $\mathcal{E}, \Gamma$. However, where the environment plays a nugatory role in the exposition of a proof, $\mathcal{E}$ will be elided in favour of just Γ. A new declaration $x : X$ can be added to the current context at any time, provided that X is already declared in Γ or $\mathcal{E}$, otherwise the addition of $x : X$ must be preceded by the addition of $X : \tau$ for some τ, e.g.

$$\Gamma, [X : U_0, x : X] .$$

If a declaration—which is often referred to as an *assumption*—is utilised by a proof in any material way, the only way that proof can remove its dependency on that assumption is by *discharging* the assumption using either the $(\rightarrow I)$ or $(\forall I)$ rules, as described later. Note that X cannot be discharged before x in the above example, because that would leave x with a dangling reference to a non-existent type.

Coq Listing 5. *(Section)* The `Section` at line 4 defines the start of a new context. The declarations at lines 1 and 5, and the definitions at lines 2 and 6, are global and local respectively. The local declaration at line 5, and the local definition at line 6, are out of scope at line 9.

```
1> Variable x : nat.
2> Definition y : nat := 1.
3> Print All.
*** [ x : nat ]
y : nat
4> Section P.
5>    Variable x : nat.
6>    Let z : nat := 2.
7>    Print All.
*** [ Top.x : nat ]
y : nat
*** [x : nat]
*** [z := 2 : nat]
8> End P.
9> Print All.
*** [ x : nat ]
y : nat
```

2.6.3 Assumption Rule

The assumption rule asserts that A is provable (i.e. inhabited) in the context of Γ, if the assumption "a is an object of A" is an element of Γ for some a. The intuition behind this

rule is that since a is assumed to be an inhabitant of A in the premiss, it makes sense to use it to prove A in the conclusion, i.e.

$$\frac{a\colon A \in \Gamma}{\Gamma \vdash a\colon A}\ (Ass)\ .$$

When space is at a premium, this rule is abbreviated by

$$\Gamma \vdash a\colon A_{(Ass)}\ .$$

Coq Listing 6. *(assumption)* The goal at line 3 is `nat` and any inhabitant of `nat` would be sufficient to prove it. The variable declaration at line 2 adds an object of `nat` to the current context, which the `assumption` tactic at line 5 implicitly uses to prove the goal.

```
1> Section P.
2>    Variable x : nat.
3>    Goal nat.
4>    Proof.
5>       assumption.
6>    Qed.
7> End P.
```

2.6.4 Type Nat

The type of natural numbers.

Formation Rule (Nat)

Nat is a small type, i.e.

$$\frac{}{Nat\colon U_0}\ (Nat\,F)\ ,$$

because it houses primitive objects, not types.

Introduction Rules (Nat)

There are two introduction rules. The first rule (the base case) asserts without precondition that O is a canonical object of Nat, i.e.

$$\frac{}{\Gamma \vdash O\colon Nat}\ (Nat\,I_1)\ .$$

The letter O is chosen over the number 0 for the time being, although the two will be aligned soon enough. The second rule (the induction case) asserts that the successor of n is a canonical object of Nat, if n is a canonical object of Nat, i.e.

$$\frac{\Gamma \vdash n\colon Nat}{\Gamma \vdash S\,n\colon Nat}\ (Nat\,I_2)\ .$$

S is a function that returns the successor of a natural number. See Figure 2.2.

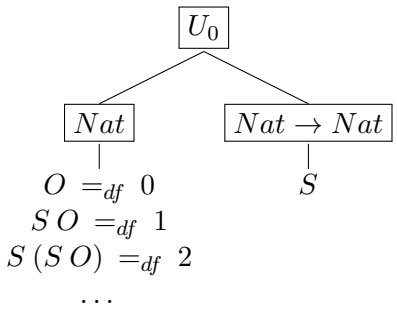

Figure 2.2: The comparative state of the system after the formation and introduction of type Nat. Note that the introduction of Nat requires the formation and introduction of type $Nat \to Nat$, to house the successor function S. See Section 2.6.9 for a description of the arrow type. In reality, the system would also include other types, e.g. a home for the destructor $@^{-1}_{Nat}$ (as defined by the elimination rule below), but these have been elided for the sake of simplicity.

Coq Listing 7. *(`nat`)* The encoding of the natural numbers in Coq is similar.

```
Inductive nat : Set :=  O : nat | S : nat -> nat
```

Example 3. Prove $\vdash S\,(S\,(S\,O))\colon Nat$, i.e. prove that $S\,(S\,(S\,O))$ is an object of type Nat.

Proof. The proof consists of a sequence of rulings (taking Γ to be the empty context in each case) in which each conclusion apart from the last is the premiss of the next ruling, i.e.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash O\colon Nat}\,(Nat\,I_1)}{\vdash S\,O\colon Nat}\,(Nat\,I_2)}{\vdash S\,(S\,O)\colon Nat}\,(Nat\,I_2)}{\vdash S\,(S\,(S\,O))\colon Nat}\,(Nat\,I_2)\,.$$

□

Coq Listing 8. *(`exact`)* In Coq, the proof is particularly simple. The goal is to find an object of `nat`. Any natural number would in fact do, but the one chosen (by means of the `exact` tactic) is the one of interest, i.e. `(S (S (S O)))`.

```
Goal nat.
Proof.
   exact (S (S (S O))).
Qed.
```

Remark 1. *(Proofs)* Three points. First, proofs are generally tree-shaped rather than chain-shaped structures (like the one in Example 3), because conclusions are invariably derived from multiple premisses. Second, there is nothing to preclude a proof from utilising the rules (introduction, elimination and so on) of many different types. Third, the proof

above is not typical of proofs in general. Usually, a proof is simply a demonstration of the fact that a type is inhabited. The inhabitant is almost always elided with the proviso that it could be determined if need be.

Elimination Rule (Nat)

The elimination rule asserts that $P\,n$ (a dependent type on Nat) is inhabited for all natural numbers n, if $P\,O$ is inhabited, and $P\,(S\,n)$ is inhabited assuming that $P\,n$ is inhabited for all natural numbers n, i.e.

$$\frac{\begin{array}{l}\Gamma \vdash P\colon Nat \to U_n \\ \Gamma \vdash i\colon P\,O \\ \Gamma \vdash j\colon \forall n\colon Nat\,.\,P\,n \to P\,(S\,n)\end{array}}{\Gamma, [n\colon Nat] \vdash @^{-1}_{Nat}\,P\,n\,i\,j\colon P\,n}\ (Nat\,E_n)\ .$$

The term $@^{-1}_{Nat}$ denotes the destructor of Nat, a function that takes four arguments P, n, i and j to produce an object of $P\,n$, where P is a function that returns a potentially different object (i.e. type) of U_n for each natural number n. Broadly speaking, this rule asserts that if a property P on n is provable every which way that n can be constructed, then it can justifiably be claimed that the property is provable for all natural numbers.

Computation Rules (Nat)

The computation rules are given by

$$\begin{aligned} @^{-1}_{Nat}\,P\,O\,i\,j &\twoheadrightarrow i \\ @^{-1}_{Nat}\,P\,(S\,n)\,i\,j &\twoheadrightarrow j\,n\,(@^{-1}_{Nat}\,P\,n\,i\,j)\ . \end{aligned}$$

Each rule asserts that the term on the right is a simplification of the term on the left, where the term on the left is a particular object of $P\,n$ for some natural number n. In the first rule, the number is O, and in the second rule the number is $S\,n$. Note that the second rule is recursive.

Example 4. Simplify

$$@^{-1}_{Nat}\,P\,(S\,(S\,O))\,i\,j\ ,$$

i.e. the object obtained by replacing n by $(S\,O)$ in the second computation rule. Applying the second rule twice and the first rule once gives

$$\begin{aligned} @^{-1}_{Nat}\,P\,(S\,(S\,O))\,i\,j &\twoheadrightarrow j\,(S\,O)\,(@^{-1}_{Nat}\,P\,(S\,O)\,i\,j) \\ &\twoheadrightarrow j\,(S\,O)\,(j\,O\,(@^{-1}_{Nat}\,P\,O\,i\,j)) \\ &\twoheadrightarrow j\,(S\,O)\,(j\,O\,i)\ . \end{aligned}$$

Now, the term $j\,(S\,O)$ can be shown to be an object of type

$$P\,(S\,O) \to P\,(S\,S\,O)\ ,$$

and the term $j\,O$ can be shown to be an object of type

$$P\,O \to P\,(S\,O)\;;$$

and together with the remaining term i, which is an object of type $P\,O$, what this computation shows is that the simplification of $@^{-1}_{Nat}P\,(S\,(S\,O))\,i\,j$—an object of type $P(S\,S\,O)$—is dependent on the computation of objects of type

$$P\,O, \quad P\,O \to P\,(S\,O), \quad P\,(S\,O) \to P\,(S\,S\,O)\;.$$

For example, using the usual symbols for the natural numbers, if

$$P \;=_{df}\; \lambda n\,.\,n \geq 0\;,$$

then proving that $2 \geq 0$ would amount to proving

$$0 \geq 0, \quad 0 \geq 0 \to 1 \geq 0, \quad 1 \geq 0 \to 2 \geq 0\;.$$

There is an affinity between the computation rules of Nat and mathematical induction, which extends to other recursively defined types as well.

Definition 17. *(Equal Nat)* Two objects n_1 and n_2 of Nat are equal if they reduce to the same canonical object, i.e.

$$n_1 \twoheadrightarrow n \twoheadleftarrow n_2$$

for some n.

Coq Listing 9. *(`nat_rect`)* The elimination and computation rules of inductive data types are automatically rolled into a function definition by Coq, which in the case of type `nat` is called `nat_rect`. Compare and contrast variables `f` and `f0` below with objects i and j above.

```
nat_rect =
fun (P : nat -> Type) (f : P 0) (f0 : forall n : nat, P n -> P (S n)) =>
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | 0 => f
  | S n0 => f0 n0 (F n0)
  end
     : forall P : nat -> Type,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

2.6.5 Type *Bool*

The type of booleans.

Formation Rule (*Bool*)

Bool is a small type, i.e.

$$\frac{}{Bool\colon U_0}\;(Bool\,F)\;.$$

Introduction Rules ($Bool$)

There are two introduction rules. The first rule asserts that $True$ is a canonical object of $Bool$, and the second rule asserts that $False$ is a canonical object of $Bool$, i.e.

$$\frac{}{\Gamma \vdash True\colon Bool}\ (Bool\, I_1) \qquad \frac{}{\Gamma \vdash False\colon Bool}\ (Bool\, I_2)\ .$$

See Figure 2.3.

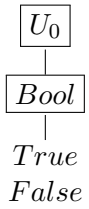

Figure 2.3: The comparative state of the system after the formation and introduction of type $Bool$.

Coq Listing 10. *(`bool`)* In Coq, the encoding of `bool` is similiar.

```
Inductive bool : Set :=  true : bool | false : bool
```

Elimination Rule ($Bool$)

The elimination rule asserts that $P\, b$ (a dependent type on $Bool$) is inhabited for all booleans b, if $P\, True$ and $P\, False$ are inhabited, i.e.

$$\frac{\begin{array}{l}\Gamma \vdash P\colon Bool \to U_n \\ \Gamma \vdash i\colon P\, True \\ \Gamma \vdash j\colon P\, False\end{array}}{\Gamma, [b\colon Bool] \vdash @^{-1}_{Bool}\, P\, b\, i\, j\colon P\, b}\ (Bool\, E_n)\ .$$

An alternative rendition of the conclusion, which is more intuitive, is

$$\Gamma, [b\colon Bool] \vdash \mathit{if}\ b\ \mathit{then}\ i\ \mathit{else}\ j\colon P\, b\ .$$

Computation Rules ($Bool$)

The computation rules are given by

$$\begin{aligned} @^{-1}_{Bool}\, P\, True\, i\, j &\twoheadrightarrow i \\ @^{-1}_{Bool}\, P\, False\, i\, j &\twoheadrightarrow j\ , \end{aligned}$$

or alternatively by

$$\begin{aligned} \mathit{if}\ True\ \mathit{then}\ i\ \mathit{else}\ j &\twoheadrightarrow i \\ \mathit{if}\ False\ \mathit{then}\ i\ \mathit{else}\ j &\twoheadrightarrow j\ . \end{aligned}$$

Definition 18. *(Equal Bool)* Two objects b_1 and b_2 of *Bool* are equal if they reduce to the same canonical object, i.e.

$$b_1 \twoheadrightarrow b \twoheadleftarrow b_2 \ ,$$

where b is either $True$ or $False$.

2.6.6 Type $a_1 =_A a_2$ or $I(A, a_1, a_2)$

The type of objects which prove that a_1 and a_2 are equal objects of A. Recall that to know a type is to know how to construct its canonical objects, and—more importantly in this context—what it means for two objects to be equal.

Formation Rule ($=_A$)

The formation rule asserts that $a_1 =_A a_2$ is an object of U_p, if A is an object of U_p, and a_1 and a_2 are objects of A, i.e.

$$\frac{A\colon U_p \quad a_1\colon A \quad a_2\colon A}{a_1 =_A a_2\colon U_p}\ (I\,F)\ .$$

See Figure 2.4. Note that $a_1 =_A a_2$ is a dependent type on A because its inhabitants vary with the values of a_1 and a_2.

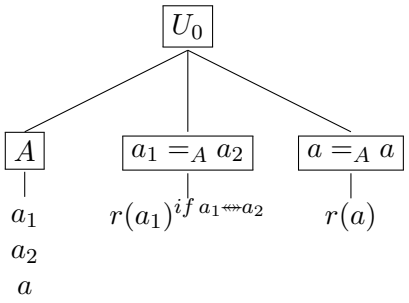

Figure 2.4: The comparative state of the system after the formation and introduction of type $a_1 =_A a_2$, when $p = 0$.

Introduction Rule ($=_A$)

There are two introduction rules. The first rule asserts that $r(a)$ is an object of $a =_A a$, if a is an object of A, i.e.

$$\frac{\Gamma \vdash a\colon A}{\Gamma \vdash r(a)\colon a =_A a}\ (II)\ .$$

The letter r stands for reflexive, because "equals" is a reflexive relation. The second rule, which is a stronger version of equality than the first rule, asserts that $r(a_1)$ is an object of $a_1 =_A a_2$, if a_1 and a_2 (both objects of A) are convertible, i.e.

$$\frac{\Gamma \vdash a_1\colon A \quad \Gamma \vdash a_2\colon A \quad \Gamma \vdash a_1 \twoheadleftarrow\twoheadrightarrow a_2}{\Gamma \vdash r(a_1)\colon a_1 =_A a_2}\ (II')\ .$$

Why $r(a_1)$ and not $r(a_2)$? No reason apart from the fact a_1 comes first. In any case, a_1 and a_2 are equal. The reader may wonder what use $r()$ is? Well, its mere presence allows things to be done which would not be so easy otherwise.

Coq Listing 11. *(eq_refl)* At line 1, the type of object `eq_refl 0` is requested, where `eq_refl 0` is Coq's notation for $r(0)$. Now, $r(0)$ is an object of $0 =_{Nat} 0$ by the (II) rule, and indeed that is what Coq displays.

```
1> Check eq_refl 0.
eq_refl 0
     : 0 = 0
2> Goal plus 1 2 = mult 1 3.
3> Proof.
4>    simpl.
5>    exact (eq_refl 3).
6> Qed.
```

At line 2, the proposition $1 + 2 =_{Nat} 1 \times 3$ is asserted. Now

$$1 + 2 \twoheadrightarrow 3 \twoheadleftarrow 1 \times 3 \,.$$

$\therefore$

$$\frac{1 + 2 \colon Nat \quad 1 \times 3 \colon Nat \quad 1 + 2 \leftrightarrows 1 \times 3}{r(1 + 2) \colon 1 + 2 =_{Nat} 1 \times 3} \; (II') \,,$$

i.e. $r(3)$ proves the proposition. At line 4, the `simpl` tactic reduces the current goal to `3 = 3` (not shown). Finally, at line 5, the `exact` tactic asserts that `eq_refl 3` is a proof of `3 = 3`. Note that the tactics in lines 4 and 5 could be replaced by `reflexivity`, a tactic which both simplifies and asserts equality.

Elimination Rule ($=_A$)

Let P be an expression in which x is a free variable. The elimination rule asserts that $J(p, r)$ is an object of $P\,[a_2/x]$, if p is an object of $P\,[a_1/x]$, and r is an object of $a_1 =_A a_2$, i.e.

$$\frac{\Gamma \vdash p \colon P\,[a_1/x] \quad \Gamma \vdash r \colon a_1 =_A a_2}{\Gamma \vdash J(p, r) \colon P\,[a_2/x]} \; (IE) \,.$$

The intuition behind this rule is simple: if a_1 and a_2 are equal, then a_1 can be substituted for a_2, and a_2 for a_1, in any context. As Leibnitz put it, "two terms are the same if one can be substituted for the other without altering the truth of any statement". Incidentally, there is nothing to preclude substituting just *some* of the occurrences of a variable in an expression, i.e. there is no requirement to substitute them *all*.

Coq Listing 12. *(rewrite)* Lines 1 to 4 set up the premisses for an instantiation of the (IE) rule. At line 5, the goal `P a2` is asserted; and at line 6, the `rewrite` tactic substitutes `a1` for `a2` in the current goal (in virtue of the introduction of `r` at line 2), and the current goal becomes `P a1`. Finally, the `exact` tactic declares `p` to be a proof of the current goal, which of course it is at line 4.

```
1> Variables a1 a2 : nat.
2> Variable r : a1 = a2.
3> Definition P (x : nat) := x = x.
4> Variable p : P a1.
5> Goal P a2.
6>    rewrite <- r.
7>    exact p.
8> Qed.
```

Computation Rule ($=_A$)

The computation rule is given by

$$J(p, r(a)) \twoheadrightarrow p(a) \ ,$$

since the term on the left is inferred from the term on the right by the (IE) rule.

2.6.7 Type $A \wedge B$

The type of objects formed by taking the conjunction of two types. This type is a special case of type $\exists a\colon A \,.\, B\, a$, which is defined in Section 2.6.11.

Formation Rule ($\wedge$)

The formation rule asserts that $A \wedge B$ is an object of $U_{max(n,m)}$, if A is an object of U_n and B is an object of U_m, i.e.

$$\frac{A\colon U_n \quad B\colon U_m}{A \wedge B\colon U_{max(n,m)}} \ (\wedge F) \ .$$

See Figure 2.5.

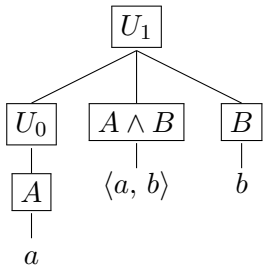

Figure 2.5: The comparative state of the system after the formation and introduction of type $A \wedge B$, when $n = 0$ and $m = 1$. The reader may wonder how B could possibly be an inhabitant of U_1? The only possible explanation is that B is a compound type, formed from the premiss of at least one object of U_1—possibly U_0 itself—before connecting with A.

Introduction Rule ($\wedge$)

The introduction rule asserts that $\langle a, b\rangle$ is an object of $A \wedge B$, if a is an object of A, and b is an object of B, i.e.

$$\frac{\Gamma \vdash a\colon A \quad \Delta \vdash b\colon B}{\Gamma, \Delta \vdash \langle a, b\rangle\colon A \wedge B}\ (\wedge I)\ .$$

For example, the objects of $Nat \wedge Bool$ consist of all possible pairs of objects of Nat and $Bool$, as shown in Figure 2.6.

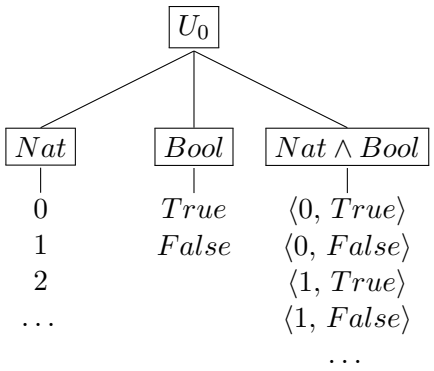

Figure 2.6: The objects $\langle n, b\rangle$ of $Nat \wedge Bool$. It would be wrong to assume that n and b need to be canonical objects of Nat and $Bool$ respectively.

If the colon relation were interpreted as "is a proof of", the introduction rule would assert that the conjunction of A and B is provable, if A and B are provable. Recall that a proposition is provable if its type is inhabited.

Coq Listing 13. *(`and`, `prod`)* In Coq, it is not possible to form the conjunction of `nat` and `bool` using type `and`, because type `and` takes arguments of `Prop`, whereas `nat` and `bool` are objects of `Set`. However, an alternative type `prod` takes objects of either `Set` or `Prop`.

```
> Print and.
Inductive and (A B : Prop) : Prop :=  conj : A -> B -> A /\ B
> Print prod.
Inductive prod (A B : Type) : Type :=  pair : A -> B -> A * B
> Check pair 0 true.
(0, true)
     : nat * bool
```

Elimination Rules ($\wedge$)

There are two elimination rules. The first rule asserts that $fst\ A\ B\ p$ is an object of A, if p is an object of $A \wedge B$, i.e.

$$\frac{\Gamma \vdash p\colon A \wedge B}{\Gamma \vdash fst\ A\ B\ p\colon A}\ (\wedge E_1)\ ,$$

where *fst* is a function that takes A, B and p to the *first* object of p. Similarly, the second rule asserts that $snd\ A\,B\,p$ is an object of B, if p is an object of $A \wedge B$, i.e.

$$\frac{\Gamma \vdash p\colon A \wedge B}{\Gamma \vdash snd\ A\,B\,p\colon B}\ (\wedge E_2)\ .$$

From the first rule, it follows that

$$fst\ Nat\ Bool\ \langle 0,\ True\rangle$$

is a non-canonical object of Nat, and from the second rule that

$$snd\ Nat\ Bool\ \langle 0,\ True\rangle$$

is a non-canonical object of $Bool$.

Coq Listing 14. *(`destruct`, `assumption`)* At line `2`, an object of `A /\ B` is declared, and at line `3`, the goal is set to prove that `A` is inhabited in the presence of `A /\ B`. At line `5`, the `destruct` tactic is employed to apply the elimination rule in reverse[6], i.e. to infer the premisses `a` and `b` (objects of `A` and `B`) from the conclusion `p`. With `a` and `b` in play, the proof concludes by asserting (with the aid of the `assumption` tactic) that the goal is proved in virtue of there being an object of `A` in existence, i.e. `a`.

```
1> Variables A B : Prop.
2> Variable p : A /\ B.
3> Goal A.
4> Proof.
5>    destruct p as [a b].
6>    assumption.
7> Qed.
```

Computation Rules ($\wedge$)

The computation rules are given by

$$fst\ A\,B\,\langle a,\ b\rangle \twoheadrightarrow a$$
$$snd\ A\,B\,\langle a,\ b\rangle \twoheadrightarrow b\ .$$

The first rule reduces a pair of objects to the first object, and the second rule does likewise for the second object. For example,

$$fst\ Nat\ Bool\ \langle 0,\ True\rangle \twoheadrightarrow 0$$
$$snd\ Nat\ Bool\ \langle 0,\ True\rangle \twoheadrightarrow True\ .$$

Coq Listing 15. *(`Eval`)* The `Eval` command normalises a term according to various reduction strategies, as defined by `compute`.

[6] Recall that Coq applies backward reasoning.

```
> Print fst.
fst =
fun (A B : Type) (p : A * B) => let (x, _) := p in x
     : forall A B : Type, A * B -> A
> Eval compute in (fst (A := nat) (B := bool) (pair 0 true)).
     = 0
     : nat
```

Definition 19. *(Equal $\wedge$)* Two objects p_1 and p_2 of type $A \wedge B$ are equal if their first and second objects are equal, i.e. if

$$\begin{aligned} \mathit{fst}\ A\ B\ p_1 &=_A \mathit{fst}\ A\ B\ p_2 \\ \mathit{snd}\ A\ B\ p_1 &=_B \mathit{snd}\ A\ B\ p_2\ . \end{aligned}$$

Remark 2. An alternative elimination rule, which is more insightful than those stated earlier, and from which those stated earlier can easily be derived, asserts that $P\ A\ B\ c$ is inhabited for all conjunctions c, if $P\ A\ B\ \langle a, b\rangle$ is inhabited for all pairs $\langle a, b\rangle$, where $a\colon A$ and $b\colon B$, i.e.

$$\frac{\begin{array}{l} P\colon \forall A\colon U_p\,.\,\forall B\colon U_q\,.\,A \wedge B \to U_n \\ A\colon U_p \\ B\colon U_q \\ c\colon A \wedge B \\ i\colon \forall A\colon U_p\,.\,\forall B\colon U_q\,.\,\forall a\colon A\,.\,\forall b\colon B\,.\,P\ A\ B\ \langle a, b\rangle \end{array}}{@_\wedge^{-1}\ P\ A\ B\ c\ i\colon P\ A\ B\ c}\ (\wedge E_n)\ .$$

The computation rule which accompanies this elimination rule is given by

$$@_\wedge^{-1}\ P\ A\ B\ \langle a, b\rangle\ i \twoheadrightarrow i\ A\ B\ a\ b\ .$$

Now, if

$$\begin{aligned} P &=_{df} \lambda A\,.\,\lambda B\,.\,\lambda c\,.\,A \\ i &=_{df} \lambda A\,.\,\lambda B\,.\,\lambda a\,.\,\lambda b\,.\,a\ , \end{aligned}$$

then

$$@_\wedge^{-1}\ P\ A\ B\ \langle a, b\rangle\ i\colon P\ A\ B\ c \twoheadrightarrow a\colon A\ ,$$

i.e. the first rule. Similarly, if

$$\begin{aligned} P &=_{df} \lambda A\,.\,\lambda B\,.\,\lambda c\,.\,B \\ i &=_{df} \lambda A\,.\,\lambda B\,.\,\lambda a\,.\,\lambda b\,.\,b\ , \end{aligned}$$

then

$$@_\wedge^{-1}\ P\ A\ B\ \langle a, b\rangle\ i\colon P\ A\ B\ c \twoheadrightarrow b\colon B\ ,$$

i.e. the second rule. Clearly, by choosing suitable values for P and i, the earlier elimination rules can be derived. This also applies to other types as well.

2.6.8 Type $A \vee B$

The type of objects formed by taking the disjunction of two types.

Formation Rule ($\vee$)

The formation rule asserts that $A \vee B$ is an object of $U_{max(n,m)}$, if A is an object of U_n and B is an object of U_m, i.e.

$$\frac{A\colon U_n \quad B\colon U_m}{A \vee B\colon U_{max(n,m)}}\ (\vee F)\ .$$

See Figure 2.7.

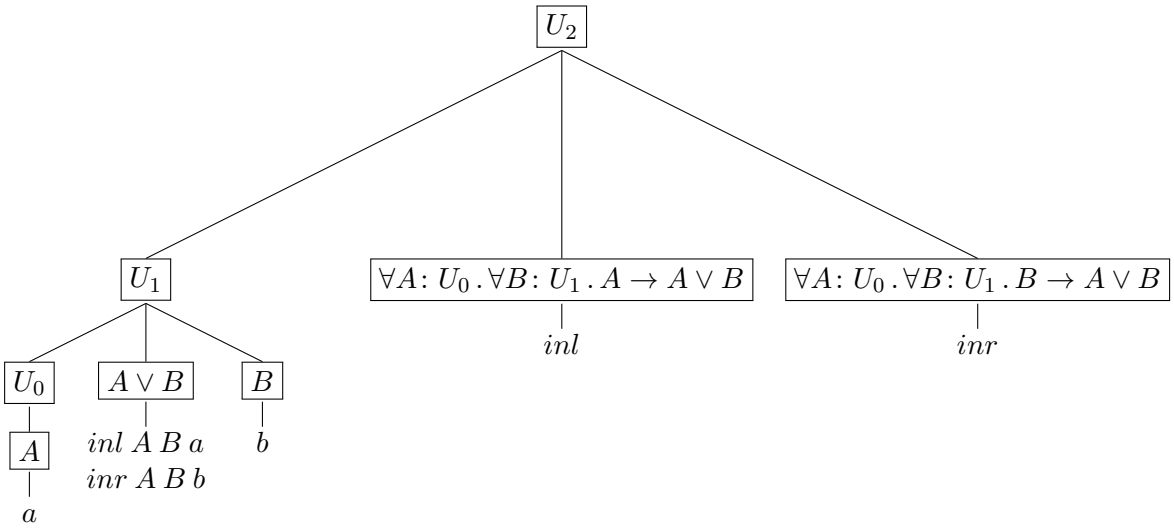

Figure 2.7: The comparative state of the system after the formation and introduction of type $A \vee B$, when $n = 0$ and $m = 1$. The $\forall a\colon A\,.\,B\ a$ type is defined in Section 2.6.10.

Introduction Rules ($\vee$)

There are two introduction rules. The first rule asserts that $inl\ A\ B\ a$ is an object of $A \vee B$, if a is an object of A, i.e.

$$\frac{\Gamma \vdash a\colon A}{\Gamma \vdash inl\ A\ B\ a\colon A \vee B}\ (\vee I_1)\ .$$

The second rule asserts that $inr\ A\ B\ b$ is an object of $A \vee B$, if b is an object of B, i.e.

$$\frac{\Gamma \vdash b\colon B}{\Gamma \vdash inr\ A\ B\ b\colon A \vee B}\ (\vee I_2)\ .$$

For inl read "in left disjunct", and for inr read "in right disjunct". An object of $A \vee B$ consists of two things: (i) an object of A or an object of B, and (ii) an indication as to whether the object is of A or B. In Section 2.3 (footnote), a proposition was proven in a classical setting by the law of the excluded middle, *without* knowing which of two contrary

disjuncts were inhabited. As stated *there*, the proof is not valid in a constructive setting, because as is plain to see *here*, it would not be possible to construct the disjunction to infer the proposition.

Coq Listing 16. *(`left, right`)* At line `2`, an object of `A` is declared, and at line `3`, the goal is set to prove that `A \/ B` is inhabited. At line `5`, the `left` tactic is employed to assert that the current goal `A \/ B` is provable in virtue of the left disjunct as opposed to the right disjunct, and the current goal therefore changes to `A`. The proof concludes with the `exact` tactic asserting that `a` inhabits the current goal, which of course it does at line `2`. Note that there is also a `right` tactic.

```
1> Variables A B : Prop.
2> Variable a : A.
3> Goal A \/ B.
4> Proof.
5>    left.
6>    exact a.
7> Qed.
```

Elimination Rule ($\vee$)

The elimination rule asserts that $P\,d$ (a dependent type on $A \vee B$) is inhabited for all disjunctions d of $A \vee B$, if $P(inl\ A\ B\ a)$ is inhabited for all objects a of A, and $P(inr\ A\ B\ b)$ is inhabited for all objects b of B, i.e.

$$\frac{\begin{array}{l} A\colon U_p \\ B\colon U_q \\ d\colon A \vee B \\ P\colon A \vee B \to U_n \\ i\colon \forall a\colon A\,.\,P\,(inl\ A\ B\ a) \\ j\colon \forall b\colon B\,.\,P\,(inr\ A\ B\ b) \end{array}}{@_{\vee}^{-1}\ A\ B\ P\ d\ i\ j\colon P\ d}\ (\vee E_n)\ .$$

Remark 3. This rule is experimental (and therefore requires further study) in that A and B are free variables in the types of P, i and j, whereas if the usual scheme had applied (as in Remark 2), A and B would have been bound. On the positive side, it simplifies the rules; on the negative side, it reduces flexibility, since A and B are obliged to take the same values throughout.

Computation Rules ($\vee$)

The computation rules are given by

$$\begin{aligned} &@_{\vee}^{-1}\ A\ B\ P\ (inl\ A\ B\ a)\ i\ j \twoheadrightarrow i\ a \\ &@_{\vee}^{-1}\ A\ B\ P\ (inr\ A\ B\ b)\ i\ j \twoheadrightarrow j\ b\ . \end{aligned}$$

Definition 20. *(Equal $\vee$)* Two objects d_1 and d_2 of $A \vee B$ are equal, that is to say the proposition $d_1 =_{A \vee B} d_2$ is inhabited, if either

$$\begin{aligned} @_{\vee}^{-1}\, A\, B\, P\, d_1\, i\, j &\twoheadrightarrow i\, a_1 \\ @_{\vee}^{-1}\, A\, B\, P\, d_2\, i\, j &\twoheadrightarrow i\, a_2 \\ a_1 &=_A a_2\ , \end{aligned}$$

or

$$\begin{aligned} @_{\vee}^{-1}\, A\, B\, P\, d_1\, i\, j &\twoheadrightarrow j\, b_1 \\ @_{\vee}^{-1}\, A\, B\, P\, d_2\, i\, j &\twoheadrightarrow j\, b_2 \\ b_1 &=_B b_2\ . \end{aligned}$$

Clearly, d_1 and d_2 could not possibly be equal unless they were both of the *inl* or *inr* variety. Further, having established that they are indeed of the same kind, showing that they are equal then amounts to demonstrating that the terms to which they compute are equal.

2.6.9 Type $A \to B$

The type of functions $\lambda a\,.\,b$ that take an object of A to an object B, where A is an object of U_p for some p, and B is an object of U_q for some q. This type is a special case of type $\forall a\colon A\,.\,B\,a$, which is defined in Section 2.6.10.

Formation Rule ($\to$)

The formation rule asserts that $A \to B$ is an object of $U_{max(p,q)}$, if A is an object of U_p, and B is an object of U_q, i.e.

$$\frac{A\colon U_p \quad B\colon U_q}{A \to B\colon U_{max(p,q)}}\ (\to F)\ .$$

See Figure 2.9.

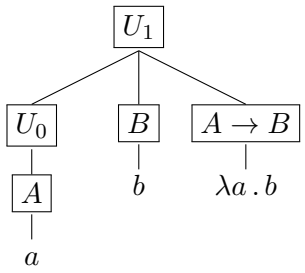

Figure 2.8: The comparative state of the system after the formation and introduction of type $A \to B$, when $p = 0$ and $q = 1$.

Introduction Rule ($\to$)

The introduction rule asserts that $\lambda a\,.\,b$ is an object of $A \to B$, if b is an object of B in a context where a is an object of A, i.e.

$$\frac{\Gamma,\, [a\colon A] \vdash b\colon B}{\Gamma \vdash \lambda a\,.\,b\colon A \to B}\ (\to I)\ .$$

Note that the $(\to I)$ rule discharges the assumption $[a\colon A]$, since it no longer appears in the context of the conclusion.

Elimination Rule ($\to$)

The elimination rule asserts that $f\,a$ is an object of B, if f is an object of $A \to B$, and a is an object of A, i.e.

$$\frac{\Gamma \vdash f\colon A \to B \quad \Gamma \vdash a\colon A}{\Gamma \vdash f\,a\colon B}\ (\to E)\ .$$

Computation Rule ($\to$)

The computation rule is given by

$$(\lambda x\,.\,b)\,a \to b\,[a/x]\ .$$

This is the β-reduction rule, as defined in Section 2.5.1.

Example 5. Prove $\vdash A \to B \to (A \wedge B)$.

Proof. The proof follows from the (Ass), $(\wedge I)$ and $(\to I)$ rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{a\colon A \in [a\colon A]}{[a\colon A] \vdash a\colon A}\ (Ass) \quad \dfrac{b\colon B \in [b\colon B]}{[b\colon B] \vdash b\colon B}\ (Ass)}{[a\colon A,\, b\colon B\,] \vdash \langle a,\, b\rangle\colon A \wedge B}\ (\wedge I)}{[a\colon A] \vdash \lambda b\,.\,\langle a,\, b\rangle\colon B \to (A \wedge B)}\ (\to I)}{\vdash \lambda a\,.\,\lambda b\,.\,\langle a,\, b\rangle\colon A \to B \to (A \wedge B)}\ (\to I)\ .$$

The proof object $\lambda a\,.\,\lambda b\,.\,\langle a,\, b\rangle$ is a function that takes an object of A and an object of B to a pair of objects of A and B. □

Coq Listing 17. *(`split`)* The Coq proof of Example 5 is shown below. `Theorem T` at line `2` names the goal so that it can be printed (for comparison purposes with the proof object in Example 5) at line `9`. The `split` tactic at line `5` splits the current goal (which at this point in the proof is `A /\ B`) into two separate goals, one for `A` and the other for `B`. Thereafter, with proofs of `A` and `B` in the context in virtue of the `intros` at line `4`, the proof follows by `assumption`.

```
1> Parameters A B : Prop.
2> Theorem T : A -> B -> (A /\ B).
3> Proof.
4>    intros a b.
5>    split.
6>    assumption
7>    assumption.
8> Qed.
9> Print T.
T = fun (a : A) (b : B) => conj a b
     : A -> B -> A /\ B
```

2.6.10 Type $\forall a\colon A\,.\,B\,a$

The type of functions $\lambda a\,.\,b$ that take an object a of A to an object b of $B\,a$ (a dependent type on A), where A is an object of U_p for some p, and B is an object of $A \to U_q$ for some q.

Formation Rule ($\forall$)

The formation rule asserts that $\forall a\colon A\,.\,B\,a$ is an object of $U_{max(p,q)}$, if A is an object of U_p, and $B\,a$ is an object of U_q in a context where a is an object of A, i.e.

$$\frac{A\colon U_p \quad [a\colon A] \vdash B\,a\colon U_q}{\forall a\colon A\,.\,B\,a\colon U_{max(p,q)}}\ (\forall F)\ .$$

See Figure 2.9.

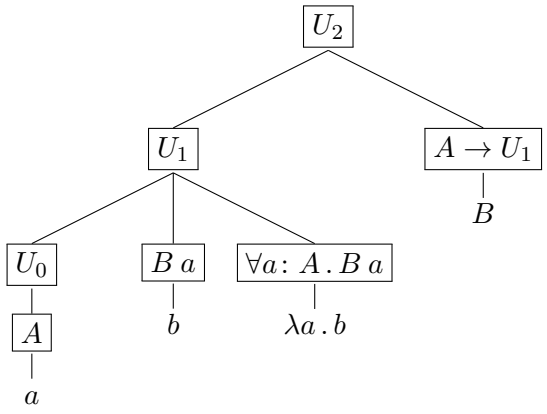

Figure 2.9: The comparative state of the system after the formation and introduction of type $\forall a\colon A\,.\,B\,a$, when $p = 0$ and $q = 1$.

Introduction Rule ($\forall$)

The introduction rule asserts that $\lambda a\,.\,b$ is an object of $\forall a\colon A\,.\,B\;a$, if b is an object of $B\;a$ in a context where a is an object of A, i.e.

$$\frac{\Gamma, [a\colon A] \vdash b\colon B\;a}{\Gamma \vdash \lambda a\,.\,b\colon \forall a\colon A\,.\,B\;a}\;(\forall I)\;.$$

Note that the $(\forall I)$ rule discharges the assumption $[a\colon A]$.

Elimination Rule ($\forall$)

The elimination rule asserts that $P\;A\;B\;f\;a$ is inhabited for all functions f of $\forall a\colon A\,.\,B\;a$, and all objects a of A, if $P\;A\;B\;(\lambda a\,.\,b)\;a$ is inhabited for all objects a of A, and all objects b of $B\;a$, i.e.

$$\frac{\begin{array}{l}P\colon \forall A\colon U_p\,.\,\forall B\colon A \to U_q\,.\,\forall a\colon A\,.\,B\;a \to U_n\\ A\colon U_p\\ B\colon A \to U_q\\ f\colon \forall a\colon A\,.\,B\;a\\ i\colon \forall A\colon U_p\,.\,\forall B\colon A \to U_q\,.\,\forall a\colon A\,.\,\forall b\colon B\;a\,.\,P\;A\;B\;(\lambda a\,.\,b)\;a\\ a\colon A\end{array}}{@_\forall^{-1}\;P\;A\;B\;f\;i\;a\colon P\;A\;B\;f\;a}\;(\forall E_n)\;.$$

The rule by which universal elimination is normally defined in text books, can be found by substituting suitable values for P, f and i in the $(\forall E_n)$ rule above. Let

$$\begin{aligned}P &=_{df} \lambda A\,.\,\lambda B\,.\,\lambda f\,.\,(\lambda a\,.\,B\;a)\\ i &=_{df} \lambda A\,.\,\lambda B\,.\,\lambda a\,.\,\lambda b\,.\,(\lambda a\,.\,b)\;a\\ f &=_{df} \lambda a\,.\,b\;.\end{aligned}$$

Substituting, simplifying and eliding where necessary gives

$$\frac{\begin{array}{l}\lambda a\,.\,b\colon \forall a\colon A\,.\,B\;a\\ a\colon A\end{array}}{(\lambda a\,.\,b)\;a\colon B\;a}\;(\forall E)\;,$$

since

$$@_\forall^{-1}\;P\;A\;B\;(\lambda a\,.\,b)\;i\;a \twoheadrightarrow i\;A\;B\;a\;b \twoheadrightarrow (\lambda a\,.\,b)\;a$$

by the computation rule below, and

$$P\;A\;B\;(\lambda a\,.\,b)\;a \twoheadrightarrow B\;a\;.$$

Putting f back and adding contexts gives

$$\frac{\begin{array}{l}\Gamma \vdash f\colon \forall a\colon A\,.\,B\;a\\ \Gamma \vdash a\colon A\end{array}}{\Gamma \vdash f\;a\colon B\;a}\;(\forall E)\;.$$

This form of the $(\forall E)$ rule, which is the one that shall henceforth be used, asserts that $f\,a$ (the application of f on a) is an object of $B\,a$, if f is an object of $\forall a\colon A\,.\,B\,a$, and a is an object of A.

Computation Rule ($\forall$)

The computation rule is given by

$$@_{\forall}^{-1}\ P\ A\ B\ (\lambda a\,.\,b)\ i\ a \twoheadrightarrow i\ A\ B\ a\ b\ .$$

Example 6. Prove $\vdash \forall n\colon Nat\,.\,n =_{Nat} n$.

Proof. The proof follows from the (Ass), (II) and $(\forall I)$ rules.

$$\dfrac{\dfrac{\dfrac{n\colon Nat \in [n\colon Nat]}{[n\colon Nat] \vdash n\colon Nat}\ (Ass)}{[n\colon Nat] \vdash r(n)\colon n =_{Nat} n}\ (II)}{\vdash \lambda n\,.\,r(n)\colon \forall n\colon Nat\,.\,n =_{Nat} n}\ (\forall I)\ .$$

The proof object $\lambda n\,.\,r(n)$ is a function that takes a natural number n to a proof $r(n)$ that n equals itself. If this specific use of the $(\forall I)$ rule is compared with its definition given earlier, it should be clear that A is Nat, $p = 0$, B is $\lambda n\,.\,n =_{Nat} n$, $q = 0$, a is n, and b is $r(n)$. □

Coq Listing 18. *(`intro`)* The Coq proof of Example 6 is shown below. First, the `intro` tactic strips aways the universal quantifier and replaces the current goal with `n = n`, then the `reflexivity` tactic asserts that `n = n` is inhabited.

```
1> Goal forall n : nat, n = n.
2> Proof.
3>    intro.
4>    reflexivity.
5> Qed.
```

2.6.11 Type $\exists a\colon A\,.\,B\,a$

The type of pairs of objects $\langle \overline{a},\, b\rangle$ in which $\overline{a}$ is an object of A, and b is an object of $B\,\overline{a}$ (a dependent type on A), where A is an object of U_p for some p, and B is an object of $A \to U_q$ for some q.[7]

Formation Rule ($\exists$)

The formation rule asserts that $\exists a\colon A\,.\,B\,a$ is an object of $U_{max(p,q)}$, if A is an object of U_p, and $B\,a$ is an object of U_q in a context where a is an object of A, i.e.

$$\dfrac{A\colon U_p \quad [a\colon A] \vdash B\,a\colon U_q}{\exists a\colon A\,.\,B\,a\colon U_{max(p,q)}}\ (\exists\, F)\ .$$

See Figure 2.10.

[7] The overline symbol denotes an existential witness.

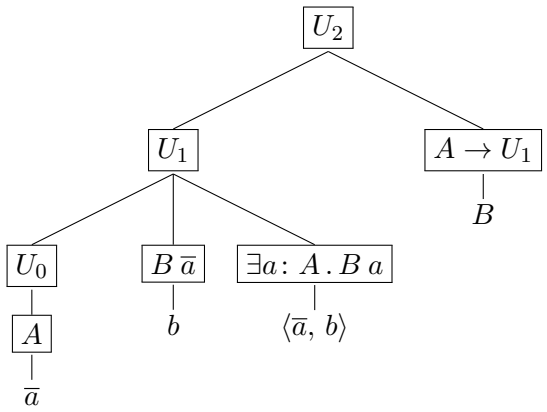

Figure 2.10: The comparative state of the system after the formation and introduction of type $\exists a\colon A\,.\,B\,a$, when $p = 0$ and $q = 1$.

Introduction Rule ($\exists$)

The introduction rule asserts that $\langle \overline{a},\, b\rangle$ is an object of $\exists a\colon A\,.\,B\,a$, if $\overline{a}$ is an object of A, and b is an object of $B\,\overline{a}$, i.e.

$$\frac{\begin{array}{c}\Gamma \vdash \overline{a}\colon A \\ \Gamma \vdash b\colon B\,\overline{a}\end{array}}{\Gamma \vdash \langle \overline{a},\, b\rangle\colon \exists a\colon A\,.\,B\,a}\ (\exists\, I)\ .$$

$\overline{a}$ is said to bear witness that b inhabits $B\,\overline{a}$.

Elimination Rule ($\exists$)

The elimination rule asserts that $P\ A\,B\,e$ is inhabited for all pairs e of $\exists a\colon A\,.\,B\,a$, if $P A\ B\ \langle a,\, b\rangle$ is inhabited for all objects a of A and b of B, i.e.

$$\frac{\begin{array}{l}P\colon \forall A\colon U_p\,.\,\forall B\colon A \to U_q\,.\,\exists a\colon A\,.\,B\,a \to U_n \\ A\colon U_p \\ B\colon A \to U_q \\ e\colon \exists a\colon A\,.\,B\,a \\ i\colon \forall A\colon U_p\,.\,\forall B\colon A \to U_q\,.\,\forall a\colon A\,.\,\forall b\colon B\,a\,.\,P\ A\ B\ \langle a,\, b\rangle\end{array}}{@_{\exists}^{-1}\ P\ A\ B\ e\ i\colon P\ A\ B\ e}\ (\exists\, E_n)\ .$$

The rules $(\exists\, E_1)$ and $(\exists\, E_2)$ by which existential elimination is defined in most text books (for example, see page 74 of [133]), can be derived from the $(\exists\, E_n)$ rule by substituting suitable values for P, e and i.

1. To derive the $(\exists\, E_1)$ rule, let

$$P_1 =_{df} \lambda A\,.\,\lambda B\,.\,\lambda e\,.\,A$$

$$\begin{aligned} i_1 &=_{df} \lambda A\,.\,\lambda B\,.\,\lambda a\,.\,\lambda b\,.\,a \\ e &=_{df} \langle \overline{a},\, b\rangle\ . \end{aligned}$$

Substituting, simplifying and eliding where necessary gives

$$\frac{\langle \overline{a},\, b\rangle \colon \exists a \colon A\,.\,B\,a}{\overline{a} \colon A}\ (\exists\, E_1)\ ,$$

since

$$@_{\exists}^{-1}\ P_1\ A\ B\ \langle \overline{a},\, b\rangle\ i_1 \twoheadrightarrow i_1\ A\ B\ \overline{a}\ b \twoheadrightarrow \overline{a}$$

by the computation rule below, and

$$P_1\ A\ B\ \langle \overline{a},\, b\rangle \twoheadrightarrow A\ .$$

Putting e back, adding contexts, and utilising the fact that $\overline{a}$ is the first component of e, gives

$$\frac{\Gamma \vdash e \colon \exists a \colon A\,.\,B\,a}{\Gamma \vdash \mathit{fst}\ A\ B\ e \colon A}\ (\exists\, E_1)\ .$$

2. Similarly, to derive the $(\exists\, E_2)$ rule, let

$$\begin{aligned} P_2 &=_{df} \lambda A\,.\,\lambda B\,.\,\lambda e\,.\,B\ (@_{\exists}^{-1}\ P_1\ A\ B\ e\ i_1) \\ i_2 &=_{df} \lambda A\,.\,\lambda B\,.\,\lambda a\,.\,\lambda b\,.\,b \\ e &=_{df} \langle \overline{a},\, b\rangle\ . \end{aligned}$$

Again, substituting, simplifying and eliding where necessary gives

$$\frac{\langle \overline{a},\, b\rangle \colon \exists a \colon A\,.\,B\,a}{b \colon B\,\overline{a}}\ (\exists\, E_2)\ ,$$

since

$$@_{\exists}^{-1}\ P_2\ A\ B\ \langle \overline{a},\, b\rangle\ i_2 \twoheadrightarrow i_2\ A\ B\ \overline{a}\ b \twoheadrightarrow b\ ,$$

and

$$P_2\ A\ B\ \langle \overline{a},\, b\rangle \twoheadrightarrow B\ (@_{\exists}^{-1}\ P_1\ A\ B\ \langle \overline{a},\, b\rangle\ i_1) \twoheadrightarrow B\,\overline{a}\ .$$

Putting e back, adding contexts, and utilising the fact that b is the second object of e, and that the type of b depends on the first object of e, gives

$$\frac{\Gamma \vdash e \colon \exists a \colon A\,.\,B\,a}{\Gamma \vdash \mathit{snd}\ A\ B\ e \colon B\ (\mathit{fst}\ A\ B\ e)}\ (\exists\, E_2)\ .$$

Computation Rule ($\exists$)

The computation rule is given by

$$@_{\exists}^{-1}\, P\, A\, B\, \langle a,\, b\rangle\, i \twoheadrightarrow i\, A\, B\, a\, b\,.$$

Example 7. Prove $\vdash \exists n\colon Nat\,.\, n =_{Nat} 0$.

Proof. The proof follows immediately from the $(\exists\, I)$ rule, i.e.

$$\frac{\vdash 0\colon Nat \qquad \vdash r(0)\colon 0 =_{Nat} 0}{\vdash \langle 0,\, r(0)\rangle\colon \exists n\colon Nat\,.\, n =_{Nat} 0}\ (\exists\, I)\ ,$$

since $r(0)$ is a proof of $0 =_{Nat} 0$. The proof object $\langle 0,\, r(0)\rangle$ is a pair, of which the first object bears witness to the fact that the second object inhabits $n =_{Nat} 0$ for some n, that n of course being 0. If this specific usage of the $(\exists\, I)$ rule is compared with the definition given earlier, clearly A is Nat, $p = 0$, B is $\lambda n\,.\, n =_{Nat} 0$, $q = 0$, $\overline{a}$ is 0, and b is $r(0)$. □

Coq Listing 19. *(exists)* The Coq proof of Example 7 is shown below. At line 3, the `exists` tactic asserts that the proposition `n = 0` is provable when `n` is replaced by `0`. Coq responds by replacing the current goal with `0 = 0` (not shown). Then at line 4, the `exact` tactic asserts that `eq_refl 0` is a proof of `0 = 0`, where `eq_refl` is the constructor of an equality type `eq` as shown.

```
1> Goal exists n : nat, n = 0.
2> Proof.
3>    exists 0.
4>    exact (eq_refl 0).
5> Qed.
6> Print eq_refl.
Inductive eq (A : Type) (x : A) : A -> Prop :=  eq_refl : x = x
```

2.6.12 Type $\top$

The one-object type, and representative of a true proposition. The intuition behind $\top$ is that since the proposition is true for obvious reasons, there is no need for more than one trivial proof.

Formation Rule ($\top$)

The formation rule asserts that $\top$ is a small type, i.e.

$$\frac{}{\top\colon U_0}\ (\top\, F)\ .$$

Introduction Rule ($\top$)

The introduction rule asserts that $Triv$ is an object of $\top$, i.e.

$$\frac{}{Triv\colon \top}\ (\top\, I)\ .$$

Coq Listing 20. *(`True`)* In Coq, the one-object type is called `True` (this should not be confused with the object `true` of `bool`), and its one and only object is called `I`.

```
> Print True.
Inductive True : Prop :=  I : True
```

Elimination Rules ($\top$)

The elimination rule, which admittedly is not very useful, asserts that Pt (a dependent type on $\top$) is inhabited for all proofs t of $\top$, if $PTriv$ is inhabited, i.e.

$$\frac{\begin{array}{l}\Gamma \vdash P\colon \top \to U_n \\ \Gamma \vdash i\colon P\, Triv\end{array}}{\Gamma, [t\colon \top] \vdash @_{\top}^{-1}\, P\, t\, i\colon P\, t}\ (\top\, E_n)\ .$$

Computation Rule ($\top$)

The computation rule—which again is not very useful—is given by

$$@_{\top}^{-1}\, P\, Triv\, i \twoheadrightarrow i\ .$$

2.6.13 Type $\bot$

The empty type, and representative of a false proposition.

Formation Rule ($\bot$)

The formation rule asserts that $\bot$ is a small type, i.e.

$$\frac{}{\bot\colon U_0}\ (\bot\, F)\ .$$

Introduction Rule ($\bot$)

There are no inhabitants of $\bot$ and therefore no introduction rules.

Coq Listing 21. *(`False`)* In Coq, the empty type is called `False`, the false or absurd proposition. This should not be confused with the object `false` of `bool`.

```
> Print False.
Inductive False : Prop :=
```

Elimination Rule ($\bot$)

The elimination rule asserts that anything holds if an object of $\bot$ is discovered, i.e.

$$\frac{p\colon \bot}{abort\ A\ p\colon A}\ (\bot E)\ ,$$

where A is representative of *any* type.

Computation Rule ($\bot$)

There is no computation rule.

Negation

In classical mathematics, where propositions are considered to be either true or false, the negation of a proposition is defined to be true if the proposition is false, and false if the proposition is true. The disjunction of a proposition A and its negation $\neg A$ is therefore true for all A, i.e.

$$\frac{}{A \vee \neg A}\ (EM)\ .$$

This is the law of the excluded middle.

In constructive mathematics, where a proposition is considered to be true if and only if its type is inhabited, the negation of a proposition is defined to be a function that takes a proof of the proposition to $\bot$, the intuition being that if a proof of the proposition leads to a contradiction, the proposition must be false.

Coq Listing 22. *(`not`)* In Coq, negation is defined by the function `not`.

```
> Print not.
not = fun A : Prop => A -> False
     : Prop -> Prop
```

Finally, the law of double negation, which classically speaking infers the truth of a proposition from the truth of the negation of the negation of the proposition, i.e.

$$\frac{\neg\neg A}{A}\ (DN)\ ,$$

which in a constructive setting is not valid.

3 Models

The author is old enough to have been around before the dawn of the software modelling era, before the word "model" entered the lexicon of software engineering. Back in those days, programming—as it was called then—was an art form rather than a scientific discipline, characterised by "large doses of folklore, black magic and occasional flashes of intuition" [141]. Unsurprisingly, it was a state of affairs that became untenable when systems began to grow in size and complexity. This chapter starts out by reflecting on some of the attempts that have been made over the years to manage the complexity of software systems, for it is undoubtedly the case that the "rise of the model" is inextricably linked with the need to manage greater and greater levels of complexity. It continues by overviewing some of the most important standards in the field—those produced by the Object Management Group (OMG)—as a means of introducing the reader to the terminology of models and transformations. However, the main purpose of this chapter is to formalise the representation of a particular kind of model—the class model—in constructive type theory.

3.1 Managing Complexity

In researching the material for this section, the author was amazed to discover how so many of the ideas of the likes of DeMarco [46], Jackson [69], Yourdon [141], Booch [19], Jacobsen [70], Rumbaugh [121], Shlaer and Mellor [128, 129] to name but a few—famous methodologists and truly remarkable figures of their time—still resonate today. In reading their most significant works, what comes across loud and clear is that they were all motivated by one thing: the desire to manage the complexity of software systems more effectively.

The story starts with the humble subroutine. As Jackson [69] writes, "the invention of the subroutine in 1949 allowed larger functions to be specified in terms of smaller functions", and argued that it was only "a relatively short step to view a program or system as having a single function that could be successively decomposed into smaller and smaller functions, until the functions at the level of the machine were reached". This is essentially the view on which functional decomposition is based, and the one underlying DeMarco's Structured Analysis and System Specification [46]. Plauger notes in the foreword, "it is a pleasure to watch the emergence of a new discipline". In DeMarco's view, a system specification consists of three things. First, a set of *data flow diagrams* comprising data flows, processes (also known as "bubbles"), data stores and sinks, for "showing the major decomposition of function and the interfaces among the pieces". Second, a *data dictionary* documenting the

interface flows and the data stores on all data flow diagrams. Third, a *transform description* describing the internals of each process in a rigorous fashion. The key goal of structured analysis is to define a useful partitioning of the system, where the rule of thumb is that a data flow diagram should not be too large to fit on a page.[1]

Yourdon [141] reported that the uptake of structured programming in the mid-1970s led to an order of magnitude improvement in the productivity, reliability and maintainability of software systems, but warned that a perfectly structured goto-less program is of little value if its basic design is unsound. Yourdon's approach was to design a system "whose pieces are small, easily related to the application, and relatively independent of each other". He proposed two criteria for judging the "goodness" of a design. First, *coupling*: a measure of the degree to which components are dependent on each other. Second, *cohesion*: a measure of the relatedness of the internals of a component. These criteria apply equally well to the construction of systems based on models: a good design is one in which there is low coupling between models and high cohesion within models.

Jackson [69] came up with a rather revolutionary idea for the time, that of relegating the consideration of a system's functions to a later step in the life cycle, promoting instead "the activity of modelling the real world", with the system's functions being "built upon a simulation of the real world" at a later stage. Jackson's model of the real world and his system's functions are not too dissimilar from the notions of platform independent and specific models of today; and his view of a model as the embodiment of a user's view of the real world, which is "more stable than the system's functions", is the reason why platform independent modelling, in particular, is so important.

Booch [19] argues that software "in the large" is inherently complex, and often exceeds the intellectual capacity of humans to comprehend. As he sees it, the task of a software development team is to engineer the illusion of simplicity.[2] Booch makes a compelling case for viewing the world as "a meaningful collection of objects that collaborate to achieve some higher level behaviour", as a means of managing the complexity of "industrial strength applications that exhibit a rich set of behaviours, tend to have a long life span, and many users rely on to work properly". Amusingly, he notes that there will always be geniuses around that can master anything, but that "there is no reason to suppose that the software engineering community has an inordinately large number of them". He goes on to state that "we cannot always rely on divine intervention to carry us through", and that "we must consider more disciplined ways to master complexity". Booch was a keen advocate of object-oriented modelling, and played a huge part in defining the Unified Modelling Language (UML) [100, 101], as described in the next section.

[1] The magical number of bubbles per page is considered to be 5 ± 2!

[2] There is an amusing cartoon in Booch [19]. There are two images. In the first image, a woman is seen approaching a rudimentary hole-in-the-wall machine. The machine has a screen and one large button, but nothing else; it could not be simpler. In the second image, the internals of the machine have spilled out onto the pavement, revealing all manner of pistons, pipes, cogs and wheels, and what looks like string holding them all together.

3.2 Object Management Group

The Object Management Group (OMG) is an "international, not-for-profit computer industry standards consortium", whose mission is to develop, with the help of its members, "enterprise integration standards that provide real-world value" [4]. The OMG was founded in 1989 by 11 companies, including Hewlett Packard, IBM, Sun Microsystems and Apple, and now boasts a membership of over 800. At its founding, the OMG set out to create a standard—the Common Object Request Broker Architecture (CORBA)—to allow software components written in different languages and running on different computers to communicate with each other. Since then, it has branched out into many other fields, including software modelling where it has chalked up a number of successes, most notably the Unified Modelling Language (UML) and the Model Driven Architecture (MDA) standards. There now follows a brief review of the most important standards.

3.2.1 Unified Modelling Language (UML)

It is a remarkable fact that between 1989 and 1994, the number of objected oriented methods in common usage increased from less than 10 to more than 50. Many users—the author included—found the task of choosing one method over another bewildering. By the mid 1990s, however, a critical mass of ideas began to form around the methods of Booch, Rumbaugh and Jacobson, and since they were already sharing each other's ideas, the three amigos—as they were often called—decided to unify their methods to help bring stability to the market place. Many companies—including the one that eventually employed them all—saw this move as strategically important to their businesses. The work to unify the methods began in earnest in 1994 when first Rumbaugh and then Jacobson joined Booch at Rational (now IBM). When the initial version was released the following year—which interestingly was a unified *language*, not a unified method—many other people and companies got involved and contributed ideas to further its development. The results of these endeavours were eventually offered to the OMG for standardisation in 1997.

The Unified Modelling Language (UML) is "a graphical language for visualising, specifying, constructing and documenting the artefacts of a software-intensive system" [20]. Such systems are built from the following kinds of building blocks.[3]

- Things
  - Structural
    - Class, Interface, Collaboration, Use Case, Active Class, Component, Node
  - Behavioural
    - Interaction, State Machine
  - Grouping
    - Package
  - Annotational

[3]This list was drawn from a relatively old UML user guide [20]. However, it is still relevant.

    * Note

- Relationships
  - Dependency, Association, Generalisation, Realisation
- Diagrams
  - Class, Object, Use Case, Sequence, Collaboration, State Chart, Activity, Component, Deployment

Of all these building blocks, only a small number are pertinent to this thesis, i.e. Class, Association, Generalisation, Class Diagram and Object Diagram; the others will not be mentioned again.

Class

A class is a description of a set of objects which share the same attributes and relationships, where an attribute is a named property of a class. Graphically speaking, an object and a class are usually rendered as named rectangles with internal compartments for attributes and operations. However, for ease of drawing the rectangles are elided in this thesis, as in Figure 3.1.

$$\begin{array}{c} \underline{\quad A \quad} \\ A_1 \colon Nat \\ A_2 \colon Bool \end{array} \qquad\qquad \begin{array}{c} \underline{\quad a \colon A \quad} \\ A_1 = 0 \colon Nat \\ A_2 = True \colon Bool \end{array}$$

Figure 3.1: A class A with attributes A_1 and A_2, and an object a of A.

At any given moment, an object of a class has a specific value assigned to each attribute of the class, where the value of an attribute is drawn from a ground type like Nat.

Association

An association is a structural relationship which describes the kinds of links that can exist between the objects of classes. Graphically speaking, an association is rendered as a solid line, possibly directed, frequently labeled (usually by R_n for some n in this thesis), and sometimes adorned by a multiplicity and a role name at one or both ends of the line, see Figure 3.2.

$$A \underset{0..*}{\overset{R_1}{\longrightarrow}} B \xrightarrow{R_2} C$$

Figure 3.2: A bidirectional many-valued association R_1 between A and B, and a unidirectional one-valued association R_2 between B and C. If a multiplicity is unspecified, it is assumed to be 1, as at A and C.

Generalisation

A generalisation is a relationship between one class—the super class—and one or more other classes—the subclasses—in which each object of the superclass is linked to an object of exactly one of the subclasses. The subclasses are *specialisations* of the superclass, and the superclass is a *generalisation* of the subclasses. Graphically speaking, a generalisation is rendered as a solid line from subclass to superclass with a curved arrowhead pointing to the superclass, as in Figure 3.3.

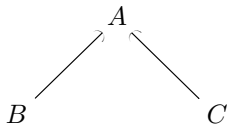

Figure 3.3: Specialisations B and C of generalisation A.

Class Diagram

A class diagram is a graphical representation of a set of classes, attributes and relationships, which captures the structural composition of a system. In most applications, this view is considered to be the most important.

Object Diagram

An object diagram is a graphical representation of a set of objects, attribute values and links, which captures a snapshot of the state of a system.

3.2.2 Object Constraint Language (OCL)

A diagram is an efficient way of conveying meaning. However, there are many nuances of meaning such as uniqueness, limits, constraints and so on, which a diagram cannot easily convey. A wholly graphical means of constructing a precise and unambiguous description of a model is simply not viable. The Object Constraint Language (OCL) [102, 139]—which has moved on from being just a constraint language since its inception—is a fully fledged object expression language, specifically designed to support the construction of *precise* UML models. It is not a stand-alone language, though, because OCL expressions cannot reside outside the context of a UML model. The power of OCL is roughly equivalent to that of first-order predicate logic.

Example 8. *(OCL)* If A is related to B via R_1 as in Figure 3.4, the following OCL

$$A \xrightarrow{R_1} \frac{B}{B_1 : Nat}$$

Figure 3.4: A simple class diagram to illustrate the evaluation of an OCL expression.

expression evaluates to true, if for all objects of B linked to `self` (an object of A) via R_1, the value of attribute B_1 exceeds 0.

```
context: A
inv: self.R_1->forAll(B_1() > 0)
```

3.2.3 Meta Object Facility (MOF)

Metadata is undoubtedly data, since it can be stored and managed in a repository. However, it is not the kind of data that can be recognised as such just by looking at it. Rather, it is the kind of data that is used to describe the meaning of other data in some context; this is what marks it out as metadata. If P and Q are data and P is known to describe Q, then P is metadata. However, unless and until the descriptive relation between P and Q is established, P is merely data. The need for applications to understand the meaning of each other's data goes back a long way. If A wishes to send data to B, then either B needs to have a built-in understanding of the structure of A's data, or A needs to tell B about it as part of its transmission. Either way, if A's data is proprietary in format, B cannot easily turn to C for an alternative source of data.[4]

The Meta Object Facility (MOF) is "an abstract language and a framework for specifying, constructing and managing technology-neutral metamodels" [97]. In the parlance of modelling, the MOF is a meta-metamodel, where a metamodel is a language for describing models. The MOF is based on a layered architecture M_0 to M_3, where the elements in any given layer describe the elements in the layer below, see Figure 3.5.

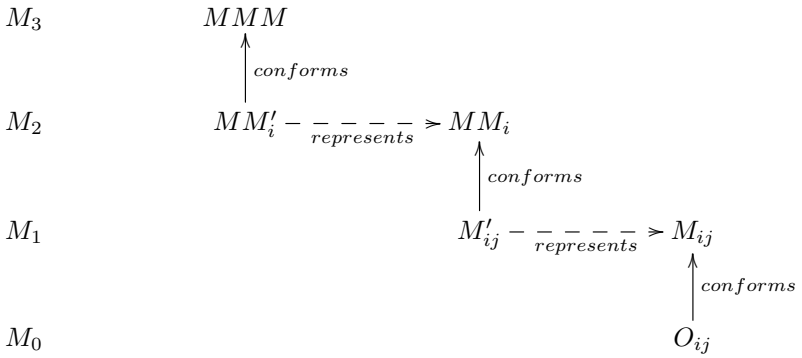

Figure 3.5: The relationships between the four layers of the MOF.

At M_3, the meta-metamodel MMM is a language that describes the elements of all metamodels MM_i at M_2. Similarly, the metamodel MM_i at M_2 for some i, is a language that describes the elements of all models M_{ij} at M_1. Lastly, the model M_{ij} at M_1 for some i and j, is a language that describes the elements of all object models O_{ij} at M_0. At M_1, there are two representations of models, a class-based representation M_{ij}, and an object-based representation M'_{ij}, where M'_{ij} plays a dual role in this framework in that it not only *conforms* to MM_i but it also *represents* M_{ij}. Similarly for MM'_i at M_2.

[4]The term *metadata* was coined by Bagley [12].

Example 9. *(MOF)* In Figure 3.6, the object model at M_0 (bottom) *conforms* to the class model at M_1 (top). Further, the object model at M_1 in Figure 3.8 both *conforms* to the metamodel at M_2 in Figure 3.7, and *represents* the class model at M_1 in Figure 3.6. It would be a simple matter to extend this example with an object model at M_2 and the meta-metamodel at M_3.

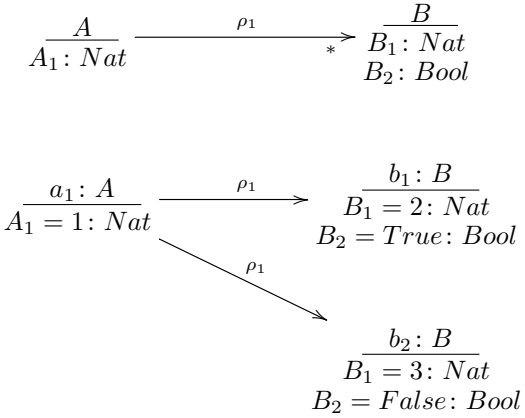

Figure 3.6: An object model at M_0 (bottom), and the corresponding class model at M_1 (top).

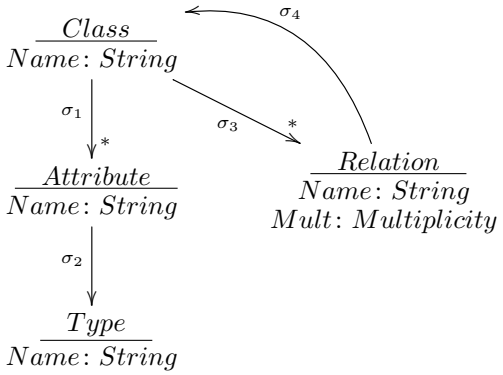

Figure 3.7: A class model at M_2.

Remark 4. *(Name Attributes)* The ubiquitous use of attributes of type string in the MOF, to uniquely identify the names of model elements at the next layer down, is unsatisfactory from a type-theoretic perspective. An experimental formalisation of a metamodel with strong types is given in Section 3.5.1.

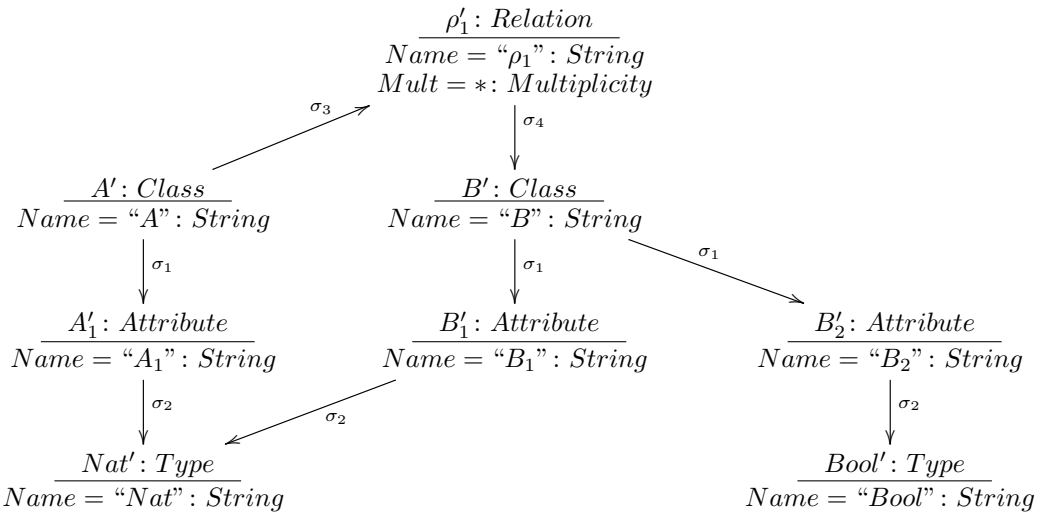

Figure 3.8: This object model at M_1 conforms to the class model at M_2 in Figure 3.7.

3.2.4 Model Driven Architecture (MDA)

As remarked earlier, the OMG was originally set up to issue CORBA-related standards within the context of the vision set out by the Object Management Architecture (OMA). However, the emergence of new kinds of standards like the UML and the MOF, which were more foundational in nature than those that preceded them, prompted the OMG to expand its vision of the OMA. This it duly did in 2001 in a paper entitled Model Driven Architecture (MDA) [96].

The MDA's approach to the specification of software systems is based on the separation of two quite distinct concerns: first, the specification of the *functionality* of a system; and second, the specification of the *implementation* of that functionality on a particular software and/or hardware infrastructure or *platform*. To this end, the MDA defines an architecture and a set of guidelines for structuring the specifications of systems based on models, where a model is "a representation of [all or] part of the function, structure and/or behaviour of a system".

Broadly speaking, there are two kinds of models in MDA: platform independent and platform specific. A *platform independent model* (PIM) is "a formal specification of the structure and function of a system that abstracts away [i.e. suppresses irrelevant] technical detail", whereas a *platform specific model* is a specification of the structure and function of a system *in the language* of the target platform. Abstracting out the fundamental structure and behaviour of a system into a PIM has three advantages. First, it makes it easier to validate the correctness of a model, since it is uncluttered by platform specific considerations. Second, it is easier to produce implementations for different platforms. Third, it aids interoperability.

Recognising that models are first class entities gives rise to the need to be able to transform them. A model transformation is a function that maps the elements of a source model to the elements of a target model according to some criteria. For example, a PIM to

PIM transformation is one that takes a source PIM to a target PIM by adding, deleting or changing elements of a platform independent nature, whereas a PIM to PSM transformation is one that typically prepares a PIM for execution on a target platform, by adding platform specific details.

3.2.5 Query/View/Transformation (QVT)

In 2002, the OMG issued a request [98] for interested parties to submit proposals for a *declarative* language that would facilitate the means to query, view and transform models. There were eight submissions in all and after an initial round of rejections, those that remained were amalgamated into a single proposal, which the OMG later adopted as the Query/View/Transformation (QVT) language [63]. The QVT language is actually comprised of three languages: relations, core and operational mappings. The relations and core languages are both declarative in nature while the operational mappings language is imperative.

- The *relations* language is used to define a transformation between two models by means of a set of relations over the elements of two metamodels, which can either be used to check that the models are consistent, or to enforce the consistency of the target model with respect to the source model, by modifying the target model as necessary. A *checked* transformation is bidirectional, whereas an *enforced* transformation is unidirectional. In the example below (a unidirectional transformation in virtue of the keyword `enforce`), a relation is defined between the class `Class` of the `UML` metamodel and the class `Table` of the `RDBMS` metamodel. The `where` clause defines the conditions under which a class `c` is related a table `t`.

```
transformation uml2rdms (uml : UML, rdbms : RDBMS){
   relation AttributeToColumn {
      checkonly domain uml c:Class {...};
      enforce domain rdbms t:Table {...};
      where {
         PrimitiveAttributeToColumn(c, t);
         ComplexAttributeToColumn(c, t);
         SuperAttributeToColumn(c, t);
      }
   }
...
}
```

- The *core* language supports the same functionality as the relations language, albeit at a lower level of abstraction. Transformations written in the core language are therefore normally much longer. The *raison d'être* for the core language appears to be that it provides a convenient way of defining the semantics of the relations language, by means of a transformation between relations and core.

- The *operational mappings* language can either be used to complement a transformation which is defined in the relations language, thereby creating a hybrid transformation, or to define a wholly imperative transformation. An example of the latter from [63] is reproduced below.

```
metamodel BOOK {
   class Book {
      title: String; composes chapters: Chapter [*]; }
   class Chapter {
      title : String; nbPages : Integer; }
}
metamodel PUB {
   class Publication {
      title : String; nbPages : Integer; }
}
transformation Book2Publication(in bookModel : BOOK, out pubModel : PUB);
main() {
   bookModel->objectsOfType(Book)->map book_to_publication();
}
mapping Class::book_to_publication() : Publication {
   title := self.title;
   nbPages := self.chapters->nbPages->sum();
}
```

3.3 Formal Class Models

As remarked earlier, the main purpose of this chapter is to formalise the representation of a class model in constructive type theory, and that is what this section discusses. The formalisation of classes, attributes, relations and generalisations below, is based on the type system defined in Section 2.6, and is written in an informative style to help readers who are unfamiliar with the subject matter.

3.3.1 Classes

This section formalises the representation of a class and an object. In summary:

Definition 21. *(Formalised Class)* A class is formalised as a small type. Recall that U_0 is the universe of small types, and so every class is formalised as an object of U_0.

Definition 22. *(Formalised Object)* An object of a class is formalised as an object of the type of the class.

Class A

Refer to Figure 3.9, which shows a class A and two of its objects a_1 and a_2. Since A is a class, A is an object of U_0, i.e. $A : U_0$. Moreover, since a_1 and a_2 are objects of class A, a_1 and a_2 are objects of type A, i.e. $a_1 : A$ and $a_2 : A$. These are the *declarations* of a_1 and a_2; their *definitions* are discussed below.

$$\frac{}{\begin{array}{l}A_1 \colon Nat \\ A_2 \colon Bool\end{array}} \qquad \frac{a_1 \colon A}{\begin{array}{l}A_1 = 1 \colon Nat \\ A_2 = True \colon Bool\end{array}} \qquad \frac{a_2 \colon A}{\begin{array}{l}A_1 = 2 \colon Nat \\ A_2 = False \colon Bool\end{array}}$$

Figure 3.9: A class A and two of its objects.

Remark 5. Two things may cause confusion in this section. First, the word "object" is used in the context of both classes and types. Second, the symbol A is simultaneously used to refer to a class and a type.

Clearly, an object of class A is constructed from a natural number and a boolean value, and an object of type A is constructed in a similar manner. Define the symbol $@_A$ to denote the constructor of an object of type A. What is the type of $@_A$? Answer: The type of function that takes an object of Nat and an object of $Bool$ to an object of A, i.e. $Nat \to Bool \to A$. So, $@_A \colon Nat \to Bool \to A$. This is the declaration of $@_A$. Now, unlike a_1 and a_2 (which are defined below), $@_A$ does not have a definition. All one can say about $@_A$ is that if it were passed an object n of Nat and an object b of $Bool$, it would return an object of A. Which object would that be? Answer: $@_A\, n\, b$. Two applications of the $(\to E)$ rule are required to confirm this.

$$\frac{\dfrac{@_A \colon Nat \to Bool \to A \quad n \colon Nat}{@_A\, n \colon Bool \to A}\,(\to E) \qquad b \colon Bool}{@_A\, n\, b \colon A}\,(\to E)\,.$$

Substituting 1 for n and $True$ for b gives the definition of a_1, i.e. $(a_1 =_{df} @_A\, 1\, True) \colon A$. Similarly, the definition of a_2 is $(a_2 =_{df} @_A\, 2\, False) \colon A$.

Coq Listing 23. *(Class A)* In Coq, a class is encoded as an inductive data type of sort `Set`.

```
Inductive A : Set :=
   Build_A : nat -> bool -> A.
Definition a1 : A :=
   Build_A 1 true.
Definition a2 : A :=
   Build_A 2 false.
```

In the previous paragraph, the type of $@_A$ (namely $Nat \to Bool \to A$) was introduced without explanation. However, being a type, it is subject to the same rules of formation, introduction and so on, as any other type. How was it formed? Answer: by applying the $(\to F)$ rule, as defined in Section 2.6.9, and utilising the fact that its constituents Nat, $Bool$ and A are all small types, i.e.

$$\frac{Nat \colon U_0 \quad \dfrac{Bool \colon U_0 \quad A \colon U_0}{Bool \to A \colon U_{max(0,0)=0}}\,(\to F)}{Nat \to Bool \to A \colon U_{max(0,0)=0}}\,(\to F)\,.$$

Clearly, $Nat \to Bool \to A$ is a small type too. Figure 3.10 gives a visualisation of the comparative state of the type system after the formalisation of class A.

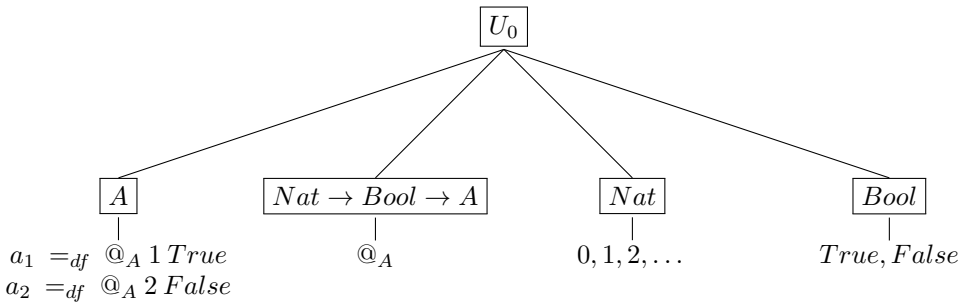

Figure 3.10: The comparative state of the type system, after the formalisation of class A, but before the formalisation of attributes A_1 and A_2.

3.3.2 Attributes

This section formalises the representation of an attribute. In summary:

Definition 23. *(Formalised Attribute)* An attribute of a class is formalised as a function that takes an object of the type of the class to an object of the type of the attribute.

Attributes A_1 and A_2

Refer to Figure 3.9 again. A_1 is formalised as an object of type $A \to Nat$, that is to say its declaration is $A_1 \colon A \to Nat$. Regarding its definition, A_1 the function must be capable of returning the value of A_1 the attribute given an arbitrary object of A, and so $A_1\ (@_A\, n\, b)$ must compute to n. Similarly, $A_2 \colon A \to Bool$, and $A_2\ (@_A\, n\, b)$ must compute to b.

A_1 and A_2 are actually defined in terms of a third function, $@_A^{-1}$, so that

$$\begin{aligned} A_1 &=_{df} \lambda a \,.\, @_A^{-1}\, \overline{p}_1\, a \\ A_2 &=_{df} \lambda a \,.\, @_A^{-1}\, \overline{p}_2\, a \end{aligned}$$

for some lists of parameters $\overline{p}_1$ and $\overline{p}_2$. The requirements on the destructor of A, as $@_A^{-1}$ is known, are interesting in that $@_A^{-1}$ must be capable of influencing both the value and the type of the term returned by $@_A^{-1}\, \overline{p}_i\, (@_A\, n\, b)$, so that when $i = 1$, the value returned is n and the type is Nat, and when $i = 2$, the value returned is b and the type is $Bool$.

Define the type of $@_A^{-1}$ to be

$$\forall P \colon A \to U_0 \,.\, [\forall n \colon Nat \,.\, \forall b \colon Bool \,.\, P\ (@_A\, n\, b)] \to \forall a \colon A \,.\, P\, a\ .$$

Admittedly, this is a complex type but it breaks down easily enough. It starts with a universal quantification over P of type $A \to U_0$, which suggests that the first parameter passed to $@_A^{-1}$ must be a function that takes an object of A to an object of U_0. For A_1, such a function is $\lambda a \,.\, Nat$. Applying the $(\forall E)$ rule gives

$$\frac{\begin{array}{l} @_A^{-1} \colon \forall P \colon A \to U_0 \,.\, [\forall n \colon Nat \,.\, \forall b \colon Bool \,.\, P\ (@_A\, n\, b)] \to \forall a \colon A \,.\, P\, a \\ \lambda a \,.\, Nat \colon A \to U_0 \end{array}}{@_A^{-1}\ (\lambda a \,.\, Nat) \colon [\forall n \colon Nat \,.\, \forall b \colon Bool \,.\, \underbrace{(\lambda a \,.\, Nat)\ (@_A\, n\, b)}_{\twoheadrightarrow Nat}] \to \forall a \colon A \,.\, \underbrace{(\lambda a \,.\, Nat)\ a}_{\twoheadrightarrow Nat}}\ (\forall E)\ .$$

The two occurrences of P in the conclusion have been replaced by $\lambda a\,.\,Nat$, as is required by the $(\forall E)$ rule. Since $\lambda a\,.\,Nat$ is a constant function, the braced terms reduce to Nat as indicated. The new type starts with an implicant, namely $\forall n\colon Nat\,.\,\forall b\colon Bool\,.\,Nat$, which suggests that the second parameter passed to $@_A^{-1}$ must be a function that takes objects of Nat and $Bool$ to Nat. For A_1, such a function is $\lambda n\,.\,\lambda b\,.\,n$. Applying the $(\to E)$ rule gives

$$\frac{\begin{array}{l}@_A^{-1}\,(\lambda a\,.\,Nat)\colon (\forall n\colon Nat\,.\,\forall b\colon Bool\,.\,Nat)\to\forall a\colon A\,.\,Nat\\ \lambda n\,.\,\lambda b\,.\,n\colon \forall n\colon Nat\,.\,\forall b\colon Bool\,.\,Nat\end{array}}{@_A^{-1}\,(\lambda a\,.\,Nat)\,(\lambda n\,.\,\lambda b\,.\,n)\colon \forall a\colon A\,.\,Nat}\;(\to E)\;.$$

This new type is a quantification over A, which suggests that the third and final parameter passed to $@_A^{-1}$ must be an object of A. Assume that a is this object. Applying the $(\forall E)$ rule to eliminate the quantifier, and then the $(\to I)$ rule to discharge the assumption, finally gives the value of A_1.

$$\underbrace{\dfrac{\dfrac{\begin{array}{l}@_A^{-1}\,(\lambda a\,.\,Nat)\,(\lambda n\,.\,\lambda b\,.\,n)\colon \forall a\colon A\,.\,Nat\\ [a\colon A]\vdash a\colon A_{(Ass)}\end{array}}{[a\colon A]\vdash @_A^{-1}\,(\lambda a\,.\,Nat)\,(\lambda n\,.\,\lambda b\,.\,n)\,a\colon Nat}\;(\forall E)\;.}{\lambda a\,.\,@_A^{-1}\,(\lambda a\,.\,Nat)\,(\lambda n\,.\,\lambda b\,.\,n)\,a\colon A\to Nat}\;(\to I)\;.}_{A_1}$$

Similarly,

$$\underbrace{\lambda a\,.\,@_A^{-1}\,(\lambda a\,.\,Bool)\,(\lambda n\,.\,\lambda b\,.\,b)\,a}_{A_2}\colon A\to Bool\;.$$

These are the definitions of A_1 and A_2.

Coq Listing 24. *(Attribute A_1)* In the following listing, there are two definitions of A_1. The first definition—which is for illustrative purposes only—is the mirror image of the one developed above (Coq calls the destructor `A_rect`). The second definition, which is more compact than the first, uses pattern matching to extract `n`.

```
Definition A1 (a : A) : nat :=
   A_rect (fun a : A => nat) (fun n : nat => fun b : bool => n) a.
Definition A1' (a : A) : nat :=
   match a with
      Build_A n b => n
   end.
Compute A1 a1.
     = 1
     : nat
Compute A1' a1.
     = 1
     : nat
```

Clearly, by passing different parameters to $@_A^{-1}$, different results can be obtained. In general,

$$@_A^{-1}\,P\,i\,a\colon P\,a\;,$$

where $P\colon A \to U_0$, $i\colon [\forall n\colon Nat\,.\,\forall b\colon Bool\,.\,P\,(@_A\,n\,b)]$, and $a\colon A$.

Finally, in order for A_1 and A_2 to be able to compute values, the behaviour of $@_A^{-1}$ must be specified. Define

$$@_A^{-1}\,P\,i\,(@_A\,n\,b) \twoheadrightarrow i\,n\,b\;.$$

The intuition behind this computation is that the left hand side computes its value by applying i to n and b. Now, in the case of A_1, i ignores b and returns n since i is $\lambda n\,.\,\lambda b\,.\,n$; and in the case of A_2, i ignores n and returns b since i is $\lambda n\,.\,\lambda b\,.\,b$. Clearly, the operational semantics of $@_A^{-1}$ are the same as those of a case statement in virtue of it returning n or b as appropriate.

The requirements placed on A_1 and A_2 have clearly been met, since

$$\begin{aligned} A_1\,(@_A\,n\,b) &\twoheadrightarrow @_A^{-1}\,(\lambda a\,.\,Nat)\,(\lambda n\,.\,\lambda b\,.\,n)\,(@_A\,n\,b) \\ &\twoheadrightarrow (\lambda n\,.\,\lambda b\,.\,n)\,n\,b \\ &\twoheadrightarrow n \end{aligned}$$

and $A_2\,(@_A\,n\,b) \twoheadrightarrow b$. Figure 3.11 shows the comparative state of the type system after the formalisation of A_1.

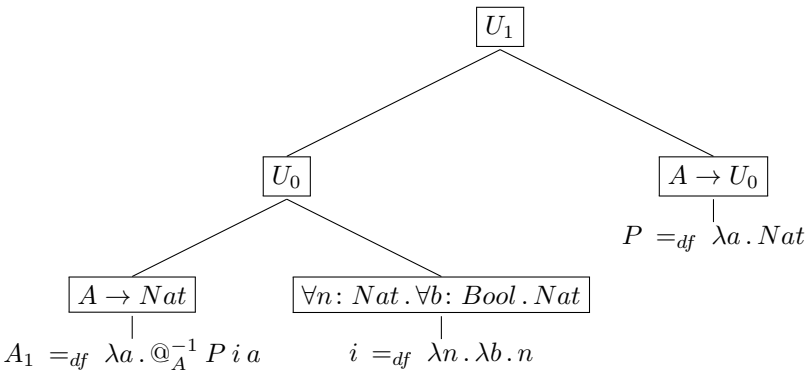

Figure 3.11: The comparative state of the type system after the formalisation of A_1. The formalisation of A_2 is similar. The destructor of A is elided for the sake of simplicity.

Before leaving this section, it is apposite to define type A in terms of its formation, introduction, elimination and computation rules, which the reader should compare and contrast with the foregoing discussion. Contrary to what one might expect, A_1 and A_2 do not feature in the definition of A because they can be derived from suitable applications of the elimination rule.

- Formation Rule

$$\frac{}{A\colon U_0}\;(A\,F)\;.$$

- Introduction Rule

$$\frac{n\colon Nat \quad b\colon Bool}{@_A\,n\,b\colon A}\;(A\,I)\;.$$

- Elimination Rule

$$\frac{\begin{array}{l} P\colon A \to U_n \\ a\colon A \\ i\colon \forall n\colon Nat\,.\,\forall b\colon Bool\,.\,P\,(@_A\,n\,b) \end{array}}{@_A^{-1}\,P\,i\,a\colon P\,a}\;(A\,E_n)\;.$$

- Computation Rule

$$@_A^{-1}\,P\,i\,(@_A\,n\,b) \twoheadrightarrow i\,n\,b\;.$$

3.3.3 Relations

This section formalises the representations of various kinds of relations. In summary:

Definition 24. *(Formalised Relation)* A unidirectional relation from a source class to a target class is formalised as a function that takes an object of the source class to either an object of the target class, an optional object of the target class, or a list of objects of the target class.

Unidirectional Relations

A unidirectional relation between a source class A and a target class B—of which there are three kinds in Figure 3.12, namely unconditional one-valued R_1, conditional one-valued R_2, and many-valued R_3—is formalised as a function that takes an object of A to an object of one of three types as follows.

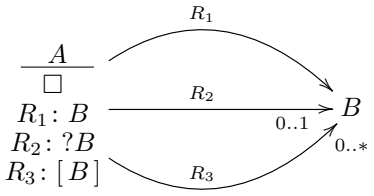

Figure 3.12: A class at the source end of three unidirectional relations.

- In the case of R_1, where the relation is unconditional and one-valued—that is to say where there is always an object of B at the end of the line—the target type is simply B, and so $R_1\colon A \to B$. If a is an object of A, then $R_1\,a$ is the related object of B by the $(\to E)$ rule, i.e.

$$\frac{R_1\colon A \to B \quad a\colon A}{R_1\,a\colon B}\;(\to E)\;.$$

  The introduction rule for A is

$$\frac{\square \quad r_1\colon B \quad r_2\colon ?B \quad r_3\colon [\,B\,]}{@_A\,\square\,r_1\,r_2\,r_3\colon A}\;(A\,I)\;,$$

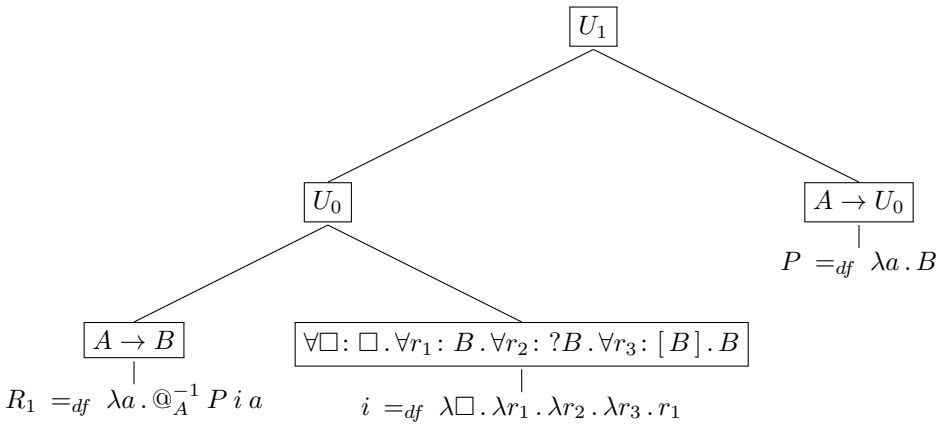

Figure 3.13: The comparative state of the type system after the formalisation of R_1. The formalisations of R_2 and R_3 are similar.

where $\Box$ is a placeholder for the elided attributes of A, and the computation rule is

$$@_A^{-1}\, P\, i\, (@_A\, \Box\, r_1\, r_2\, r_3) \twoheadrightarrow i\, \Box\, r_1\, r_2\, r_3 \;,$$

where $P\colon A \to U_0$ and

$$i\colon \forall \Box\colon \Box \,.\, \forall r_1\colon B \,.\, \forall r_2\colon ?B \,.\, \forall r_3\colon [\,B\,] \,.\, P\, (@_A\, \Box\, r_1\, r_2\, r_3) \;.$$

The requirement on R_1 is that $R_1\ (@_A\, \Box\, r_1\, r_2\, r_3) \twoheadrightarrow r_1$. In light of the detailed discussion in the previous section regarding the definitions of attributes A_1 and A_2 (admittedly, for some other class A), the requirement can be met if

$$R_1 =_{df} \lambda a \,.\, @_A^{-1}\, P\, i\, a \;,$$

where

$$\begin{aligned} P &=_{df} \lambda a \,.\, B \\ i &=_{df} \lambda \Box \,.\, \lambda r_1 \,.\, \lambda r_2 \,.\, \lambda r_3 \,.\, r_1 \;. \end{aligned}$$

Figure 3.13 shows the comparative state of the type system after the formalisation of R_1.

Coq Listing 25. *(Relation R_1)* In Coq, the definition of `B` must appear before the definition of `A` if the types are to be defined inductively. Apart from that, the definition of `R1` is much like `A1`. Note that the computation of `(R1 a1)` returns `Build_B`, not `b1`, although `b1` has the same value.

```
Inductive B : Set :=
   Build_B : B.
Inductive A : Set :=
   Build_A : B -> option B -> list B -> A.
```

```
Definition R1 (a : A) : B :=
   match a with
      Build_A r1 r2 r3 => r1
   end.
Definition b1 : B :=
   Build_B.
Definition a1 : A :=
   Build_A b1 None nil.
Compute R1 a1.
     = Build_B
     : B
```

- In the case of R_2, where the relation is conditional and one-valued—that is to say where there may or may not be an object of B at the end of the line—the target type is the option type $?B$, and so $R_2 \colon A \to ?B$. The option type $?B$ is defined below in the usual fashion.[5] However, unlike previous types, there are two introduction and two computation rules, to cover the cases where there is and is not an object of B.

  – Formation Rule

  $$\frac{}{?B \colon U_0}\ (?B\,F)\ .$$

  – Introduction Rules

  $$\frac{}{None_B \colon ?B}\ (?B\,I_1)\ . \qquad \frac{b \colon B}{Some_B\,b \colon ?B}\ (?B\,I_2)\ .$$

  – Elimination Rule

  $$\frac{\begin{array}{l} P \colon ?B \to U_n \\ ?b \colon ?B \\ i \colon P\,None_B \\ j \colon \forall b \colon B\,.\,P\,(Some_B\,b) \end{array}}{@^{-1}_{?B}\,P\,i\,j\,?b \colon P\,?b}\ (?B\,E_n)\ .$$

  – Computation Rules

  $$@^{-1}_{?B}\,P\,i\,j\,None_B \twoheadrightarrow i$$
  $$@^{-1}_{?B}\,P\,i\,j\,(Some_B\,b) \twoheadrightarrow j\,b\ .$$

  The requirement on R_2, namely that $R_2\,(@_A\,\Box\,r_1\,r_2\,r_3) \twoheadrightarrow r_2$, can be met if

  $$R_2 =_{df} \lambda a\,.\,@^{-1}_A\,P\,i\,a\ ,$$

  where $P =_{df} \lambda a\,.\,?B$ and $i =_{df} \lambda\Box\,.\,\lambda r_1\,.\,\lambda r_2\,.\,\lambda r_3\,.\,r_2$. Let $a_1 =_{df} (@_A\,\Box\,r_1\,None_B\,r_3)$ and let $a_2 =_{df} (@_A\,\Box\,r_1\,(Some_B\,b)\,r_3)$ for some r_1, r_3 and b. Clearly, a_1 is not linked to an object of B via R_2, whereas a_2 is. As required,

  $$R_2\,a_1 \twoheadrightarrow @^{-1}_A\,P\,i\,(@_A\,\Box\,r_1\,None_B\,r_3) \twoheadrightarrow None_B\ ,$$

  and $R_2\,a_2 \twoheadrightarrow Some\,b$.

[5] There is a polymorphic version of this type which obviates the need to define a separate option type for each class.

- In the case of R_3, where the relation is many-valued—that is to say where there may be many objects of B at the end of the line—the target type is the list type $[\,B\,]$, and so $R_3 \colon A \to [\,B\,]$. The list type is defined below.[6] However, unlike previous types, this type is recursive.

  – Formation Rule

  $$\frac{}{[\,B\,] \colon U_0}\ ([\,B\,]\,F)\ .$$

  – Introduction Rules

  $$\frac{}{Nil_B \colon [\,B\,]}\ ([\,B\,]\,I_1)\ . \qquad \frac{b \colon B \quad l \colon [\,B\,]}{Cons_B\, b\, l \colon [\,B\,]}\ ([\,B\,]\,I_2)\ .$$

  – Elimination Rule

  $$\frac{\begin{array}{l} P \colon [\,B\,] \to U_n \\ l \colon [\,B\,] \\ i \colon P\, Nil_B \\ j \colon \forall b \colon B\,.\, \forall l \colon [\,B\,]\,.\, P\, l \to P\, (Cons_B\, b\, l) \end{array}}{@^{-1}_{[\,B\,]}\, P\, i\, j\, l \colon P\, l}\ ([\,B\,]\,E_n)\ .$$

  – Computation Rules

  $$@^{-1}_{[\,B\,]}\, P\, i\, j\, Nil_B \twoheadrightarrow i$$
  $$@^{-1}_{[\,B\,]}\, P\, i\, j\, (Cons_B\, b\, l) \twoheadrightarrow j\, b\, l\, (@^{-1}_{[\,B\,]}\, P\, i\, j\, l)\ .$$

  The requirement on R_3, i.e. that $R_3\, (@_A\, \Box\, r_1\, r_2\, r_3) \twoheadrightarrow r_3$, can be met if

  $$R_3 =_{df} \lambda a\,.\, @^{-1}_A\, P\, i\, a\ ,$$

  where $P =_{df} \lambda a\,.\,[\,B\,]$ and $i =_{df} \lambda \Box\,.\,\lambda r_1\,.\,\lambda r_2\,.\,\lambda r_3\,.\,r_3$.

Bidirectional Relations

A bidirectional relation R_1 between A and B (see Figure 3.14) is formalised by two unidirectional relations, which can either go from A to B and from B to A, or from a manufactured third class AB, say, to A and B. The former is more compact than the latter and also more intuitive, but the latter is applicable in more situations than the former (see Section 3.4 for an explanation as to why this is the case).

[6] There is also a polymorphic version of this type.

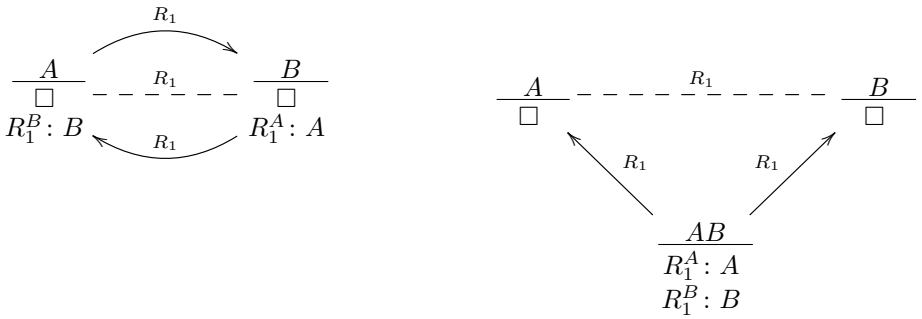

Figure 3.14: Two ways of formalising a bidirectional relation.

3.3.4 Generalisations

This section formalises the representation of a generalisation. In summary:

Definition 25. *(Formalised Generalisation)* A generalisation is formalised by $n+1$ unidirectional relations, where n is the number of subclasses. The relation from the superclass to the subclasses is an n-way disjunction, and the n relations from the subclasses to the superclass are unconditional and one-valued.

A generalisation R_1 between a superclass A and two subclasses B and C (see Figure 3.15) is formalised by three functions: a function R_1^A that takes an object of A to an object of $B \vee C$, a function R_1^B that takes an object of B to an object of A, and a function R_1^C that takes an object of C to an object of A. So, $R_1^A : A \to B \vee C$, $R_1^B : B \to A$ and $R_1^C : C \to A$.

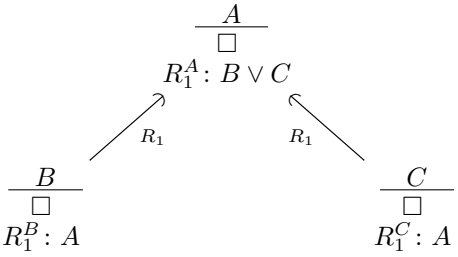

Figure 3.15: A generalisation R_1 between a superclass A and subclasses B and C.

The requirement on R_1^A, i.e. that $R_1^A\,(@_A\,\square\,r) \twoheadrightarrow r$, can be met if

$$R_1^A \;=_{df}\; \lambda a\,.\,@_A^{-1}\,P\,i\,a\;,$$

where $P \;=_{df}\; \lambda a\,.\,B \vee C$ and $i \;=_{df}\; \lambda\square\,.\,\lambda r\,.\,r$. Similarly for R_1^B and R_1^C.

3.4 Inductive, Mutually Inductive and Coinductive Models

Informally, a class model is a directed graph (C, R) of disjoints sets of classes and relations, together with two maps $init\colon R \to C$ and $term\colon R \to C$, assigning to each relation r an

initial class $init(r)$ and a terminal class $term(r)$. For example, ignoring the properties of relations, the class model in Figure 3.16 is given by the directed graph $(\{A, B, C\}, \{R_1, R_2\})$, where $init(R_1) = A$, $term(R_1) = B$, $init(R_2) = B$, and $term(R_2) = C$.

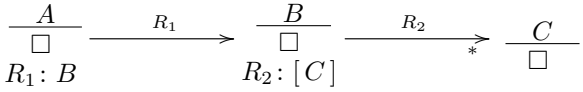

Figure 3.16: An inductive class model.

There are three kinds of class models: inductive, mutually inductive, and coinductive.

Definition 26. *(Inductive Model)* An inductive class model is one in which both the class model and the object models are *acyclic* directed graphs.

An example of an inductive class model is shown in Figure 3.16. An object model of an inductive class model is assembled by constructing—for all relations—the objects of terminal classes before the objects of initial classes. So, objects of C are constructed before objects of B, and objects of B are constructed before objects of A.

Definition 27. *(Mutually Inductive Model)* A mutually inductive class model is one in which the class model is a *cyclic* directed graph, and the object models are *acyclic* directed graphs.

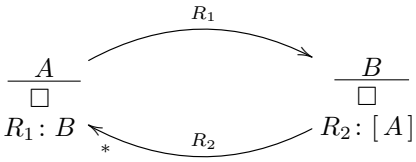

Figure 3.17: A mutually inductive class model.

An example of a mutually inductive class model is shown in Figure 3.17. A mutually inductive object model is assembled in the same way as an inductive object model. See Listing 26 and Figure 3.18 for an example.

Coq Listing 26. *(Mutually Inductive Model)* In Coq, the `with` clause is used to define a mutually inductive set of types. The order in which the objects are constructed is consistent with Figure 3.18, in that for all relations, the terminal classes are constructed before the initial classes.

```
Inductive A : Set :=
   Build_A : B -> A
with B : Set :=
   Build_B : list A -> B.
Definition b2 : B := Build_B nil.
Definition b3 : B := Build_B nil.
Definition a2 : A := Build_A b2.
```

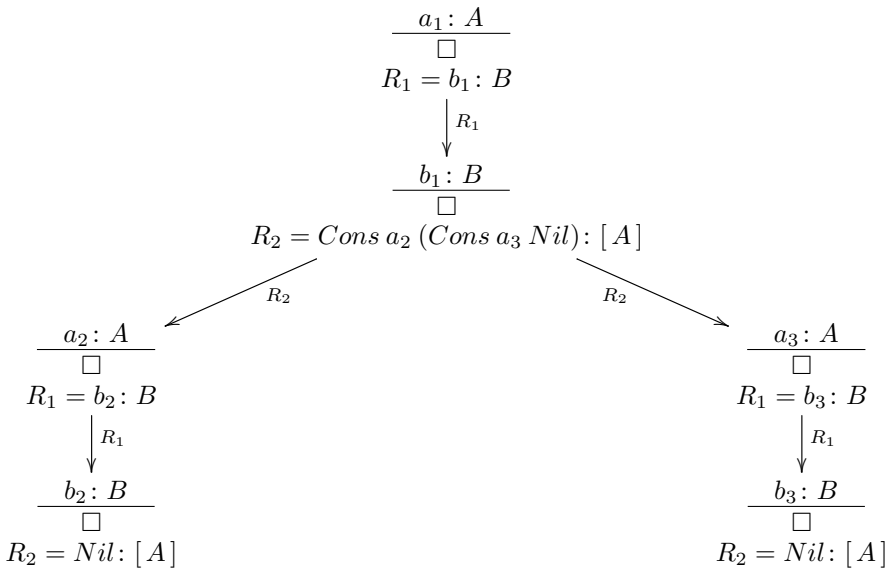

Figure 3.18: An object model conforming to the mutually inductive class model in Figure 3.17.

```
Definition a3 : A := Build_A b3.
Definition b1 : B := Build_B (a2 :: a3 :: nil).
Definition a1 : A := Build_A b1.
```

Definition 28. *(Coinductive Model)* A coinductive class model is one in which both the class model and the object models are *cyclic* directed graphs.

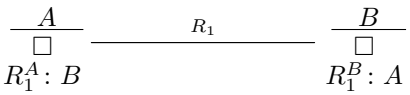

Figure 3.19: A coinductive class model.

An example of a coinductive class model is shown in Figure 3.19. A coinductive object model is assembled by means of a single—and potentially large—mutually recursive ensemble of object definitions, as shown in Listing 27 and Figure 3.20 (a more significant example can be found in Listing 34).

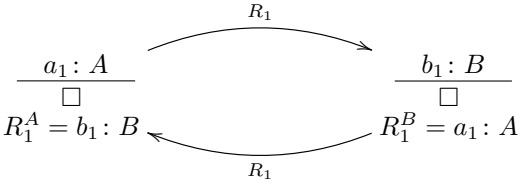

Figure 3.20: An object model conforming to the coinductive class model in Figure 3.19.

Coq Listing 27. *(Coinductive Model)* A `CoFixpoint` is used to assemble the object model of Figure 3.20.

```
CoInductive A : Set :=
   Build_A : B -> A
with B : Set :=
   Build_B : A -> B.
CoFixpoint a1 : A :=
   Build_A b1
with b1 : B :=
   Build_B a1.
```

3.5 Ordered Class Models

An ordered class model is one in which the classes are ordered by containment. To help explain what this means, this section defines such a model in three quite different ways: first in graph theory, then in set theory, and finally (as an experimental aside) in type theory. Although the graph theoretical and set theoretical definitions elide the attributes of classes and the multiplicities of relations, they are nevertheless still insightful.

Definition 29. *(Ordered Class Model - Graph)* An ordered class model (C, R) is a directed, acyclic graph of classes C and relations R, where for all classes $c \in C$ there is at most one relation $r \in R$ where $term(r) = c$.

1. It is permissible for there to exist many relations r where $init(r) = c$.
2. For all relations r, $init(r)$ is said to contain $term(r)$.
3. If there does not exist a relation r where $term(r) = c$, then c is called the *root* class of (C, R).
4. If there does not exist a relation r where $init(r) = c$, then c is called a *leaf* class of (C, R).
5. The ordered class model in Figure 3.21 is given by $(\{A, B, C, D\}, \{R_1, R_2, R_3\})$, where $init(R_1) = A$, $term(R_1) = B$, $init(R_2) = B$, $term(R_2) = C$, $init(R_3) = B$, and $term(R_1) = D$.

The following set theoretical definition has more resonance with the definition of an ordered model transformation in Chapter 5.

Definition 30. *(Ordered Class Model - Set)* An ordered class model $(C, <_C)$ is a strict partial order $<_C$ over a set of classes C.

1. If $c_1, c_2 \in C$, and $c_1 <_C c_2$, then c_2 is said to contain c_1.
2. If c is a class that contains every other class in C, either directely or indirectly, then c is called the *root* class of $(C, <_C)$.

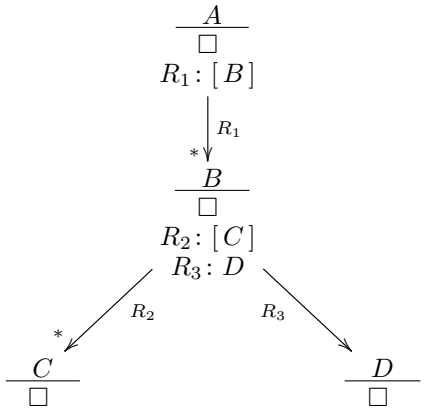

Figure 3.21: An ordered class model rooted at A.

3. If c is a class that does not contain any other class in C, then c is called a *leaf* class of $(C, <_C)$.

4. The ordered class model in Figure 3.21 is given by $(\{A, B, C, D\}, \{(C, B), (D, B), (B, A)\})$.

3.5.1 Experimental Metamodel

This section contains an experimental formalisation of a metamodel of ordered class models, as a means of answering the question, "what exactly is an ordered class model?" The graph and set theoretical definitions above are unable to answer this question because they elide certain information. As the reader will see, an ordered class model is simply an object of a particular type. Recall that a metamodel is a language for describing the structure and behaviour of particular kinds of models. In this case, the metamodel is a "model of ordered class models".

The metamodel of ordered class models is shown in Figure 3.22, and comprises five dependent types: $Model(M_o)$, $Class(C_l)$, $Attribute(A_t)$, $Type(T_y)$ and $Relation(R_e)$. Briefly, an ordered class model is rooted at a particular class; a class has zero or more attributes; an attribute has a type; a class also has zero or more relations; and a relation targets a class. Define the metamodel to be mutually inductive, so that there are no loops at the model level.

The formation and introduction rules of each type are given below. The elimination and computation rules are intentionally undefined (they serve no purpose here).

Type $Type(T_y)$ The introduction rule asserts that $@_{T_y}\ X$ is an encoding of type X.

$$\frac{X : U_0}{T_y\ X : U_1}\ (T_y\ F) \qquad \frac{X : U_0}{@_{T_y}\ X : T_y\ X}\ (T_y\ I)$$

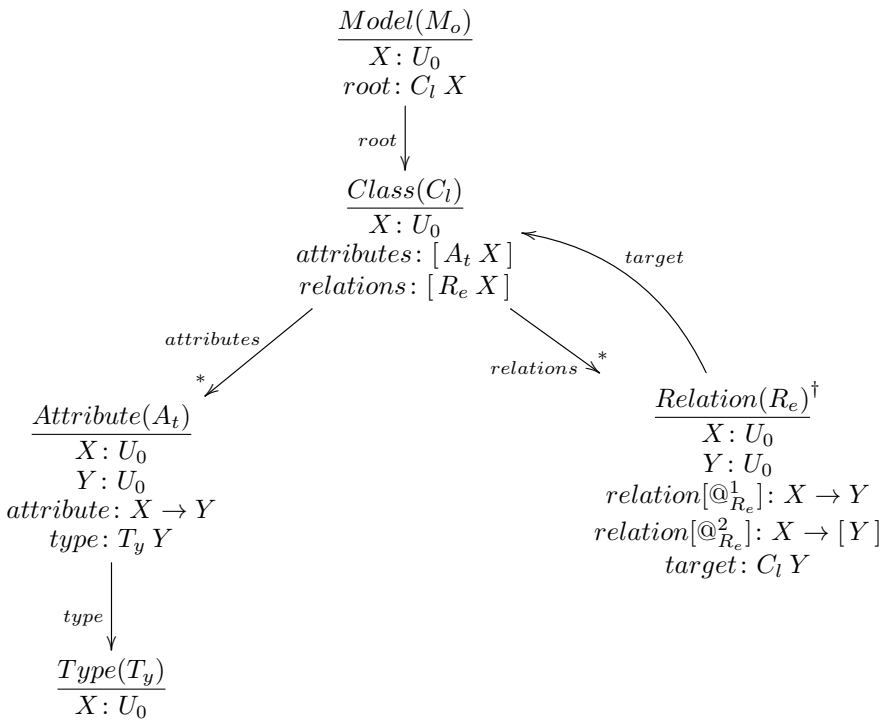

Figure 3.22: A metamodel of ordered class models.

Type *Attribute*(A_t) The introduction rule asserts that $@_{A_t}\ X\ Y\ \mathit{attribute}\ \mathit{type}$ is an encoding of an attribute of class X whose type is Y, if *attribute* is a function that takes X to Y, and *type* is an encoding of type Y.

$$\frac{X : U_0}{A_t\ X : U_1}\ (A_t\ F) \qquad \frac{X : U_0 \quad Y : U_0 \quad \mathit{attribute} : X \to Y \quad \mathit{type} : T_y\ Y}{@_{A_t}\ X\ Y\ \mathit{attribute}\ \mathit{type} : A_t\ X}\ (A_t\ I)$$

Type *Class*(C_l) The introduction rule asserts that $@_{C_l}\ X\ \mathit{attributes}\ \mathit{relations}$ is an encoding of class X, if *attributes* is a list of encodings of attributes of X, and *relations* is a list of encodings of relations originating from X.

$$\frac{X : U_0}{C_l\ X : U_1}\ (C_l\ F) \qquad \frac{X : U_0 \quad \mathit{attributes} : [\ A_t\ X\ ] \quad \mathit{relations} : [\ R_e\ X\ ]}{@_{C_l}\ X\ \mathit{attributes}\ \mathit{relations} : C_l\ X}\ (C_l\ I)$$

Type *Relation*(R_e) The first introduction rule asserts that $@^1_{R_e}\ X\ Y\ \mathit{relation}\ \mathit{target}$ is an encoding of a one-valued relation between X and Y, if *relation* is a function that takes X to Y, and *target* is an encoding of target class Y. The second introduction rule asserts some-

thing similar for many-valued relations.

$$\frac{X : U_0}{R_e\, X : U_1}\ (R_e\, F) \qquad \frac{X : U_0 \quad Y : U_0 \quad relation : X \to Y \quad target : C_l\, Y}{@^1_{R_e}\, X\, Y\, relation\ target : R_e\, X}\ (R_e\, I_1)$$

$$\frac{X : U_0 \quad Y : U_0 \quad relation : X \to [\,Y\,] \quad target : C_l\, Y}{@^2_{R_e}\, X\, Y\, relation\ target : R_e\, X}\ (R_e\, I_2)$$

Type $Model(M_o)$ The introduction rule asserts that $@_{M_o}\, X\, root$ is an encoding of an ordered class model rooted at X, if $root$ is an encoding of class X.

$$\frac{X : U_0}{M_o\, X : U_1}\ (M_o\, F) \qquad \frac{X : U_0 \quad root : C_l\, X}{@_{M_o}\, X\, root : M_o\, X}\ (M_o\, I)$$

Coq Listing 28. *(Metamodel)* Types `Ty`, `At` and `Mo` are formalised as purely inductive types, whereas `Cl` and `Re` are formalised as a mutually inductive pair of types.

```
Inductive Ty : Set -> Type :=
   Build_Ty : forall X : Set, Ty X.
Inductive At : Set -> Type :=
   Build_At : forall X : Set, forall Y : Set, (X -> Y) -> Ty Y -> At X.
Inductive Cl : Set -> Type :=
   Build_Cl : forall X : Set, list (At X) -> list (Re X) -> Cl X
with Re : Set -> Type :=
   Build_Re : forall X : Set, forall Y : Set, Cl Y -> (X -> Y) -> Re X |
   Build_Re_m : forall X : Set, forall Y : Set, Cl Y -> (X -> list Y) -> Re X.
Inductive Mo : Set -> Type :=
   Build_Mo : forall X : Set, Cl X -> Mo X.
```

$$\begin{array}{c} \underline{A} \\ A_1 : Nat \\ A_2 : Bool \\ R_1 : B \end{array} \xrightarrow{R_1} \underline{\begin{array}{c} B \\ B_1 : Nat \end{array}}$$

Figure 3.23: An ordered class model rooted at A, as encoded in Listing 29.

Coq Listing 29. *(Model)* The model in Figure 3.23 is an ordered class model because it can be encoded in the metamodel of ordered class models as follows.

```
Record B : Set := {B1 : nat}.
Record A : Set := {A1 : nat; A2 : bool; R1 : B}.
Definition E_nat : Ty nat := Build_Ty nat.
Definition E_bool : Ty bool := Build_Ty bool.
Definition E_A1 : At A := Build_At A nat A1 E_nat.
Definition E_A2 : At A := Build_At A bool A2 E_bool.
Definition E_B1 : At B := Build_At B nat B1 E_nat.
Definition E_B : Cl B := Build_Cl B (E_B1 :: nil) nil.
```

```
Definition E_R1 : Re A := Build_Re A B E_B R1.
Definition E_A : Cl A := Build_Cl A (E_A1 :: E_A2 :: nil) (E_R1 :: nil).
Definition E_AB : Mo A := Build_Mo A E_A.
```

4

Model Transformations

This chapter provides an introduction to the specification and implementation of model transformations in constructive type theory. It starts by describing the logical proposition on which all model transformations are based, i.e. the ∀∃ formula, and continues with three detailed worked examples of model transformations. It concludes by discussing the formalisation of a recursive transformation, from which the specification of an arbitrarily large ordered model transformation, as described in the next chapter, was developed.

4.1 ∀∃ Formula

This section starts by discussing the ∀∃ formula in the context of a number transformation, to familiarise readers with the way transformations are specified, implemented and certified in constructive type theory.

4.1.1 Number Transformation

The proposition $n \leq m$ asserts that either "n is less than m" or "n is equal to m"; but what exactly does this mean? Clearly, if n and m are equal, the proposition is true in virtue of the right disjunct "n is equal to m", but what if n and m are not equal? A type-theoretic approach to this problem is to associate the proposition with a type, and to define the type in such a way that the proposition is true if and only if the type is inhabited. So, "type $1 \leq 1$" is inhabited whereas "type $2 \leq 1$" is not. A suitable definition of "type $n \leq m$" is given by the following introduction rules.[1]

$$\frac{n\colon Nat}{@^1_{\leq}\, n\colon n \leq n}\ (@_{\leq} I_1)\ . \qquad \frac{n\colon Nat \quad m\colon Nat \quad h\colon n \leq m}{@^2_{\leq}\, n\, m\, h\colon n \leq S\, m}\ (@_{\leq} I_2)\ .$$

The first rule asserts that $@^1_{\leq}\, n$ is an object of type $n \leq n$, if n is a natural number, so $0 \leq 0$, $1 \leq 1$ and so on. The second rule asserts that $@^2_{\leq}\, n\, m\, h$ is an object of type $n \leq S\, m$, if n and m are natural numbers, and h is an object (a proof) that $n \leq m$. So e.g. $1 \leq 3$ if $1 \leq 2$. Recall that S is the successor function. Using these rules, it is relatively easy to

[1] Type $n \leq m$ is actually a dependent type over natural numbers, whose inhabitants vary with n and m so that, for example, the inhabitants of type $1 \leq 1$ differ from those of type $1 \leq 2$. The use of infix notation makes the types easier to read. An alternative notation, which is less readable, is given by $\leq n\, m$, or $le\, n\, m$ as in Coq.

prove that $n \leq m$ for some n and m, when n really is less than or equal to m according to these rules, by finding a suitable object that inhabits type $n \leq m$.

Example 10. Prove $\vdash 1 \leq 3$.

$$\dfrac{1\colon Nat \quad 2\colon Nat \qquad \dfrac{1\colon Nat \quad 1\colon Nat \qquad \dfrac{1\colon Nat}{@^1_{\leq}\, 1\colon 1 \leq 1}\,(@_{\leq} I_1)}{@^2_{\leq}\, 1\, 1\, (@^1_{\leq}\, 1)\colon 1 \leq 2}\,(@_{\leq} I_2)}{@^2_{\leq}\, 1\, 2\, (@^2_{\leq}\, 1\, 1\, (@^1_{\leq}\, 1))\colon 1 \leq 3}\,(@_{\leq} I_2)\ .$$

Applying the first rule once and the second rule twice gives an object of type $1 \leq 3$, i.e.

$$@^2_{\leq}\, 1\, 2\, (@^2_{\leq}\, 1\, 1\, (@^1_{\leq}\, 1))\ .$$

Since type $1 \leq 3$ is inhabited, the proposition $1 \leq 3$ is true.

Coq Listing 30. *(1 ≤ 3)* In Coq, the "less than or equal" type is defined by `le`.

```
Inductive le (n : nat) : nat -> Prop :=
    le_n : n <= n | le_S : forall m : nat, n <= m -> n <= S m
Theorem T : 1 <= 3.
Proof.
   apply le_S.
   apply le_S.
   apply le_n.
Qed.
> Print T.
T = le_S 1 2 (le_S 1 1 (le_n 1))
     : 1 <= 3
```

Now define the "less than" relation $<$ as a function that returns the "less than or equal to" type $n \leq S\, m$ for some n and m, i.e.

$$<\, =_{df}\ \lambda n\,.\,\lambda m\,.\,n \leq S\, m\colon Nat \to Nat \to U_0\ ,$$

and consider the proposition, "there is a number y such that 1 is less than y", i.e.

$$\exists y\colon Nat\,.\,1 < y\ .$$

As described in Section 2.6.11, an existential type is inhabited by pairs of elements, where the second element is a proof that the first element—the so-called witness—satisfies the proposition in question, in this case $1 < y$. Clearly, there are many possible witnesses for y: 2, 3, 4 and so on, since $1 < 2$, $1 < 3$, $1 < 4$ and so on. Reusing the proof of $1 \leq 3$ in the example above, and expanding $1 \leq 3$ to $1 < 2$ using the "less than" function, leads to a proof of the existential type as follows.

$$\dfrac{\dfrac{@^2_{\leq}\, 1\, 2\, (@^2_{\leq}\, 1\, 1\, (@^1_{\leq}\, 1))\colon 1 \leq 3}{@^2_{\leq}\, 1\, 2\, (@^2_{\leq}\, 1\, 1\, (@^1_{\leq}\, 1))\colon 1 < 2}\,(=_{exp})}{\langle 2,\, @^2_{\leq}\, 1\, 2\, (@^2_{\leq}\, 1\, 1\, (@^1_{\leq}\, 1))\rangle\colon \exists y\colon Nat\,.\,1 < y}\,(\exists\, I)\ .$$

Now consider the proposition, "there is a number y such that x is less than y for some x, i.e.

$$[x\colon Nat] \vdash \exists y\colon Nat\,.\,x < y\ .$$

Applying the $(@_{\leq} I_1)$ rule once and the $(@_{\leq} I_2)$ rule twice, and then expanding the conclusion using the "less than" function as before, leads to a proof of the hypothetical existential type, i.e.

$$\dfrac{\dfrac{\dfrac{[x\colon Nat] \vdash x\colon Nat_{(Ass)}}{[x\colon Nat] \vdash @^1_{\leq}\, x\colon x \leq x}\,(@_{\leq} I_1)}{[x\colon Nat] \vdash @^2_{\leq}\, x\, x\, (@^1_{\leq}\, x)\colon x \leq S\, x}\,(@_{\leq} I_2)}{\dfrac{[x\colon Nat] \vdash @^2_{\leq}\, x\, (S\, x)\, (@^2_{\leq}\, x\, x\, (@^1_{\leq}\, x))\colon x \leq S\, S\, x}{[x\colon Nat] \vdash @^2_{\leq}\, x\, (S\, x)\, (@^2_{\leq}\, x\, x\, (@^1_{\leq}\, x))\colon x < S\, x}\,(=_{exp})}\,(@_{\leq} I_2)}{[x\colon Nat] \vdash \langle S\, x,\, @^2_{\leq}\, x\, (S\, x)\, (@^2_{\leq}\, x\, x\, (@^1_{\leq}\, x))\rangle\colon \exists y\colon Nat\,.\,x < y}\,(\exists\, I)\ .$$

Discharging the assumption using the $(\forall I)$ rule gives

$$\dfrac{[x\colon Nat] \vdash \langle S\, x,\, @^2_{\leq}\, x\, (S\, x)\, (@^2_{\leq}\, x\, x\, (@^1_{\leq}\, x))\rangle\colon \exists y\colon Nat\,.\,x < y}{\vdash \lambda x\,.\,\langle S\, x,\, @^2_{\leq}\, x\, (S\, x)\, (@^2_{\leq}\, x\, x\, (@^1_{\leq}\, x))\rangle\colon \forall x\colon Nat\,.\,\exists y\colon Nat\,.\,x < y}\,(\forall I)\ .$$

So,

$$\forall x\colon Nat\,.\,\exists y\colon Nat\,.\,x < y$$

is inhabited by

$$\lambda x\,.\,\langle S\, x,\, @^2_{\leq}\, x\, (S\, x)\, (@^2_{\leq}\, x\, x\, (@^1_{\leq}\, x))\rangle\ ,$$

a function that takes an arbitrary object x of Nat to a corresponding object $(S\, x)$ of Nat, together with a proof

$$@^2_{\leq}\, x\, (S\, x)\, (@^2_{\leq}\, x\, x\, (@^1_{\leq}\, x))$$

that $x < S\, x$. This $\forall\exists$ type can be interpreted as the specification of a transformation between objects x and y of Nat, subject to x being less than y; and the function that inhabits this type can be viewed as a *certified program* that implements the transformation, since it not only defines the transformation function $x \mapsto S\, x$, but it also carries with it a proof of the correctness of the transformation.

Definition 31. *(Certified Program)* If f is a function that takes A to B, and p is an object of type $P(x, (f\ x))$ for some property P, then $\lambda x\,.\,\langle (f\ x), p\rangle$ is an object of type $\forall x\colon A\,.\,\exists y\colon B\,.\,P(x, y)$. Interpreting f as a *program* suggests that $\lambda x\,.\,\langle (f\ x), p\rangle$ is a *certified program*, and p is a *certificate* of its correctness.

4.1.2 Model Transformation

Turning now to model transformations, consider a transformation between two classes A and B (see Figure 4.1), in which each object x of A is transformed into a corresponding object y of B, subject to attribute B_1 of y having the same value as attribute A_1 of x. The logical interpretation of the transformation is given by

$$\forall x\colon A\,.\,\exists y\colon B\,.\,A_1\,x = B_1\,y\ ,$$

where attribute functions A_1 and B_1 are defined like others of their kind, as described in Section 3.3.

$$\frac{A}{A_1\colon Nat} \vdash\!\!-\!\!-\overset{f}{-}\!\!-\!\!\succ \frac{B}{B_1\colon Nat}$$

Figure 4.1: A simple transformation between classes A and B.

Remark 6. Not all transformations are certifiable because some types are simply not inhabited. □

Remark 7. A proof tree is drawn with its leaf terms at the top and its root term at the bottom because that is the easiest way to read it. However, that is not the easiest way to develop it. The easiest way to develop it is to start at the bottom and work up to the top (as Coq does) by applying *backward reasoning*. Applying forward reasoning requires much more insight, particularly when proofs are large. □

The following is a step by step guide to developing a certified program for the simple class to class transformation above.

1. Start with the specification of the transformation.

$$\frac{}{\forall x\colon A\,.\,\exists y\colon B\,.\,A_1\,x = B_1\,y}\ .$$

2. Apply the $(\forall I)$ rule in reverse to strip away the universal quantifier.

$$\frac{[x\colon A] \vdash \exists y\colon B\,.\,A_1\,x = B_1\,y}{\forall x\colon A\,.\,\exists y\colon B\,.\,A_1\,x = B_1\,y}\ (\forall I)\ .$$

3. Apply the $(\exists\, I)$ rule in reverse to strip away the existential quantifier. At the same time, introduce a function f of type $A \to B$, and substitute $(f\,x)$ for y in the post-condition.

$$\frac{\dfrac{[x\colon A] \vdash (A_1\,x = B_1\,y)\,[(f\,x)/y]}{[x\colon A] \vdash \exists y\colon B\,.\,A_1\,x = B_1\,y}\ (\exists\, I)}{\forall x\colon A\,.\,\exists y\colon B\,.\,A_1\,x = B_1\,y}\ (\forall I)\ .$$

4. Define f so that the new leaf term (Coq calls this the current goal) is inhabited, which in this case means defining f so that

$$(A_1\ x = B_1\ y)\,[(f\ x)/y] \twoheadrightarrow A_1\ x = A_1\ x\ .$$

Clearly, the more complex the postcondition, the more difficult this task. In this case, the task is easy. Define

$$f\ =_{df}\ \lambda x\,.\,@_B\ (A_1\ x)\ ,$$

and check that it produces the desired result, i.e.

$$(A_1\ x = B_1\ y)\,[(f\ x)/y] \twoheadrightarrow A_1\ x = B_1\ (f\ x) \twoheadrightarrow A_1\ x = A_1\ x\ .$$

5. Add the proof of $A_1\ x = A_1\ x$, i.e. $r(A_1\ x)$. Recall that equality types are inhabited by "r" terms, as described in Section 2.6.6.

$$\dfrac{\dfrac{[x\colon A] \vdash r(A_1\ x)\colon A_1\ x = A_1\ x}{[x\colon A] \vdash \exists y\colon B\,.\,A_1\ x = B_1\ y}\ (\exists\, I)}{\forall x\colon A\,.\,\exists y\colon B\,.\,A_1\ x = B_1\ y}\ (\forall I)\ .$$

6. Add the proof of the existential type as a pair, where the second element $r(A_1\ x)$ is a proof that the first element $(f\ x)$—or its normal form $@_B\ (A_1\ x)$—satisfies the postcondition.

$$\dfrac{\dfrac{[x\colon A] \vdash r(A_1\ x)\colon A_1\ x = A_1\ x}{[x\colon A] \vdash \langle @_B\ (A_1\ x),\, r(A_1\ x)\rangle\colon \exists y\colon B\,.\,A_1\ x = B_1\ y}\ (\exists\, I)}{\forall x\colon A\,.\,\exists y\colon B\,.\,A_1\ x = B_1\ y}\ (\forall I)\ .$$

7. Finally, add the proof of the universal quantifier, i.e. a function over A.

$$\dfrac{\dfrac{[x\colon A] \vdash r(A_1\ x)\colon A_1\ x = A_1\ x}{[x\colon A] \vdash \langle @_B\ (A_1\ x),\, r(A_1\ x)\rangle\colon \exists y\colon B\,.\,A_1\ x = B_1\ y}\ (\exists\, I)}{\lambda x\,.\,\langle @_B\ (A_1\ x),\, r(A_1\ x)\rangle\colon \forall x\colon A\,.\,\exists y\colon B\,.\,A_1\ x = B_1\ y}\ (\forall I)\ .$$

In summary, the program which implements the transformation is f, the certified program is $\lambda x\,.\,\langle (f\ x),\, r(A_1\ x)\rangle$, and the certificate is $r(A_1\ x)$.

Remark 8. In theory, each node of a proof tree is a term of the form $x\colon A$. However, in virtue of the close correspondence that exists between objects x and types A, courtesy of the Curry-Howard Isomorphism, there is little point in specifying both. In most of the proof trees hereafter, only the types are specified, there being an implicit understanding that the objects of those types *could* be derived in a straightforward manner if they were really required. One often delegates the task of finding the root object to a theorem prover like Coq. □

4.2 Example A

This is the first of three worked examples. Consider a transformation between two classes A and B, in which each object x of A is transformed into an object y of B, subject to three conditions (see Figure 4.2).

- The value of attribute B_1 of y is the same as the value of attribute A_1 of x.
- The value of attribute B_2 of y is the same as the value of attribute A_3 of x, if the value of attribute A_2 of x is $True$.
- The value of attribute B_2 of y is the same as the value of attribute A_4 of x, if the value of attribute A_2 of x is $False$.

$$\begin{array}{l} \underline{\quad A \quad} \\ A_1 \colon Nat \\ A_2 \colon Bool \\ A_3 \colon Nat \\ A_4 \colon Nat \end{array} \quad \overset{f}{\vdash - - \succ} \quad \begin{array}{l} \underline{\quad B \quad} \\ B_1 \colon Nat \\ B_2 \colon Nat \end{array}$$

Figure 4.2: The source and target classes of the first worked example.

The logical interpretation of the transformation is given by

$$\forall x \colon A \,.\, \exists y \colon B \,.\, P(x, y) \;, \tag{4.1}$$

where $P(x, y)$ is the conjunction of three propositions, i.e.

$$(A_1\, x = B_1\, y) \;\wedge (A_2\, x = True \to A_3\, x = B_2\, y) \;\wedge (A_2\, x = False \to A_4\, x = B_2\, y) \;.$$

The introduction rules for types A and B are as follows.

$$\frac{n_1 \colon Nat \quad b \colon Bool \quad n_2 \colon Nat \quad n_3 \colon Nat}{@_A\, n_1\, b\, n_2\, n_3 \colon A}\;(A\,I)\;. \qquad \frac{n_1 \colon Nat \quad n_2 \colon Nat}{@_B\, n_1\, n_2 \colon B}\;(B\,I)\;.$$

The elimination and computation rules are intentionally undefined. Further, the attribute functions A_1 through A_4, and B_1 and B_2, are as usual defined by the destructors $@_A^{-1}$ and $@_B^{-1}$ of A and B respectively. Four steps are required to prove that (4.1) is inhabited and that the transformation is certifiable as follows.

1. Prove
$$[x \colon A] \vdash (A_1\, x = B_1\, y)\,[(f\, x)/y] \;,$$
for a suitable function f of type $A \to B$. Now, in virtue of its type, f must be a function of the form
$$\lambda x \,.\, @_B\, n_1\, n_2$$
for some n_1 and n_2, where n_1 and n_2 determine the values assigned to attributes B_1 and B_2 respectively. Note that n_1 and n_2 both depend on x. Clearly, n_1 must be

$(A_1\ x)$, and n_2 must be an "if-then-else" term, which is conditional on the value of attribute A_2, i.e.

$$f\ =_{df}\ \lambda x\,.\,@_B\ (A_1\ x)\ (if\ (A_2\ x)\ then\ (A_3\ x)\ else\ (A_4\ x))\ .$$

The proof follows by the $(=_{exp})$ rule, since

$$(A_1\ x = B_1\ y)\,[(f\ x)/y] \twoheadrightarrow A_1\ x = B_1\ (f\ x) \twoheadrightarrow A_1\ x = A_1\ x\ ,$$

i.e.

$$\dfrac{\dfrac{[x\colon A] \vdash A_1\ x\colon Nat}{[x\colon A] \vdash A_1\ x =_{Nat} A_1\ x}\ (II)\quad \texttt{reflexivity}}{[x\colon A] \vdash (A_1\ x = B_1\ y)\,[(f\ x)/y]}\ (=_{exp})\quad \texttt{simpl}\ .$$

Disregard the Coq tactics for the time being.

2. Prove

$$[x\colon A] \vdash (A_2\ x = True \to A_3\ x = B_2\ y)\,[(f\ x)/y]\ .$$

The reader is advised to work up from the root.

$$\dfrac{\dfrac{\dfrac{\dfrac{[x\colon A] \vdash A_3\ x\colon Nat}{[x\colon A] \vdash A_3\ x =_{Nat} A_3\ x}\ (II)\quad \texttt{reflexivity}}{[x\colon A] \vdash A_3\ x = if\ True\ then\ (A_3\ x)\ else\ (A_4\ x)}\ (=_{exp})\quad \texttt{simpl}}{[x\colon A,\ h\colon A_2\ x = True\,] \vdash A_2\ x = True_{(Ass)}}\ (IE)\quad \texttt{rewrite}}{[x\colon A,\ h\colon A_2\ x = True\,] \vdash A_3\ x = if\ (A_2\ x)\ then\ (A_3\ x)\ else\ (A_4\ x)}\ (=_{exp})\quad \texttt{simpl}$$
$$\dfrac{[x\colon A,\ h\colon A_2\ x = True\,] \vdash A_3\ x = B_2\ (f\ x)}{[x\colon A] \vdash (A_2\ x = True \to A_3\ x = B_2\ y)\,[(f\ x)/y]}\ (\to I)\quad \texttt{intro}\ .$$

3. Prove

$$[x\colon A] \vdash (A_2\ x = False \to A_4\ x = B_2\ y)\,[(f\ x)/y]\ .$$

The proof is similar to step 2.

4. Finally, prove (4.1) using steps 1, 2 and 3.

$$\dfrac{\dfrac{\begin{array}{l}[x\colon A] \vdash (A_1\ x = B_1\ y)\,[(f\ x)/y] \\ [x\colon A] \vdash (A_2\ x = True \to A_3\ x = B_2\ y)\,[(f\ x)/y] \\ [x\colon A] \vdash (A_2\ x = False \to A_4\ x = B_2\ y)\,[(f\ x)/y]\end{array}}{[x\colon A] \vdash P(x,y)\,[(f\ x)/y]}\ (\wedge I)_{\times 3}\quad \texttt{split}}{[x\colon A] \vdash \exists y\colon B\,.\,P(x,y)}\ (\exists I)\quad \texttt{exists}$$
$$\dfrac{[x\colon A] \vdash \exists y\colon B\,.\,P(x,y)}{\vdash \forall x\colon A\,.\,\exists y\colon B\,.\,P(x,y)}\ (\forall I)\quad \texttt{intro}\ .$$

The certified program which implements this transformation resides in the ultimate conclusion of the proof tree, and could be determined by systematically walking through the tree from leaves to root adding objects according to the rules of type inference, as defined in Section 2.6. However, once a proof has been established, it is sensible to encode it in Coq, so that it can be managed more easily.

Coq Listing 31. *(Example A)* The proof below (the section between `Proof` and `Qed`) is not very illuminating because it fails to capture the individual effects of the tactics. Therefore, for this proof only, the tactics have placed alongside the inference rules in the hand rolled proof above, so that the correspondence between the two is clear. For example, in step 4, `intro` performs the $(\forall I)$ rule in reverse, and `exists` performs the $(\exists\, I)$ in reverse. Note that the correspondence between tactics and inference rules is not one-to-one. Finally, the certified program for this worked example is given by `T`.

```
Record A : Set := {A1 : nat; A2 : bool; A3 : nat; A4 : nat}.
Record B : Set := {B1 : nat; B2 : nat}.

Definition P (x : A) (y : B) :=
   (A1 x = B1 y) /\
   (A2 x = true -> A3 x = B2 y) /\
   (A2 x = false -> A4 x = B2 y).

Definition f (x : A) : B :=
   Build_B (A1 x) (if (A2 x) then (A3 x) else (A4 x)).

Theorem T : forall x : A, exists y : B, P x y.
Proof.
   intro.
   exists (f x).
   split.
   reflexivity.
   split.
   intro.
   simpl.
   rewrite H.
   reflexivity.
   intro.
   simpl.
   rewrite H.
   reflexivity.
Qed.

> Print T.
T =
fun x : A =>
ex_intro (fun y : B => P x y) (f x)
  (conj (eq_refl (B1 (f x)))
     (conj
        (fun H : A2 x = true =>
         eq_ind_r (fun b : bool => A3 x = (if b then A3 x else A4 x))
           (eq_refl (A3 x)) H)
        (fun H : A2 x = false =>
         eq_ind_r (fun b : bool => A4 x = (if b then A3 x else A4 x))
           (eq_refl (A4 x)) H)))
     : forall x : A, exists y : B, P x y
```

4.3 Example B

Consider a transformation between a source model AB and a target model WX (see Figure 4.3), in which each object a of A is transformed into an object w of W, and each object b of B that is linked to a via relation R_1, is transformed into an object x of X that is linked to w via relation S_1, with the additional proviso that the attribute values A_1 of a and W_1 of w, and B_1 of b and X_1 of x, are equal.

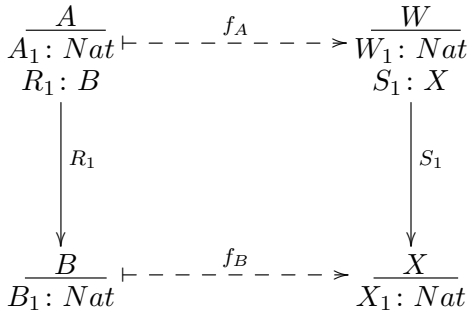

Figure 4.3: The source and target models of the second worked example.

The transformation between B and X is nested within the transformation between A and W, by means of the relations R_1 and S_1. This state of affairs is reflected in the logical interpretation of the transformation, namely

$$\begin{aligned} &\forall a\colon A\,.\,\exists w\colon W\,.\,\{^1P(a,w) \wedge \\ &\qquad \forall b\colon B\,.\,b = R_1\,a \rightarrow \exists x\colon X\,.\,\{^2Q(b,x)\ \wedge x = S_1\,w\}^2\}^1\ , \end{aligned} \tag{4.2}$$

where

$$\begin{aligned} P(a,w) &=_{df}\ A_1\,a = W_1\,w \\ Q(b,x) &=_{df}\ B_1\,b = X_1\,x\ . \end{aligned}$$

The braces, which are included for pedagogical reasons only, highlight the postconditions of the outer and inner transformations. The outer postcondition (1) is the conjunction of $P(a,w)$—a predicate describing the relation between the attribute values of corresponding objects of A and W—and the logical interpretation of the transformation between B and X, which is predicated on the assumption that B is contained by A. The inner postcondition (2) is the conjunction of $Q(b,x)$—a predicate describing the relation between the attribute values of corresponding objects of B and X—and an assertion that X is contained by W.

The introduction rules for A, B, W and X are given below. As usual, the elimination and computation rules are left undefined.

$$\frac{n\colon Nat \quad r\colon B}{@_A\,n\,r\colon A}\ (A\,I)\ . \qquad \frac{n\colon Nat}{@_B\,n\colon B}\ (B\,I)\ .$$

$$\frac{n\colon Nat \quad s\colon X}{@_W\,n\,s\colon W}\ (W\,I)\ . \qquad \frac{n\colon Nat}{@_X\,n\colon X}\ (X\,I)\ .$$

Using backward reasoning, the proof of (4.2) is given by

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta_B \vdash Q(b,x)\,[f_B\ b/x] \quad \Delta_A, \Delta_B \vdash x = S_1\ w\,[f_A\ a/w]\,[f_B\ b/x]}{\Delta_A, \Delta_B \vdash Q(b,x)\ \wedge x = S_1\ w\,[f_A\ a/w]\,[f_B\ b/x]}\ (\wedge I)}{\Delta_A, \Delta_B \vdash \exists x\colon X\,.\,Q(b,x)\ \wedge x = S_1\ w\,[f_A\ a/w]}\ (\exists I)}{\Delta_A, \Delta_{B^-} \vdash b = R_1\ a \to \exists x\colon X\,.\,Q(b,x)\ \wedge x = S_1\ w\,[f_A\ a/w]}\ (\to I)}{\dfrac{\Delta_A \vdash P(a,w)\,[f_A\ a/w] \qquad \Delta_A \vdash \forall b\colon B\,.\,b = R_1\ a \to \ldots\,[f_A\ a/w]}{\Delta_A \vdash (P(a,w)\ \wedge \forall b\colon B\,.\,b = R_1\ a \to \ldots)\,[f_A\ a/w]}\ (\wedge I)}\ (\forall I)$$

$$\dfrac{\Delta_A \vdash \exists w\colon W\,.\,P(a,w)\ \wedge \ldots}{\vdash \forall a\colon A\,.\,\exists w\colon W\,.\,P(a,w)\ \wedge \ldots}\ (\exists I)\ ,$$

where f_A and f_B are, as yet, undefined functions of type $A \to W$ and $B \to X$,

$$\begin{aligned}
&\Delta_A \vdash P(a,w)\,[f_A\ a/w]\\
&\Delta_B \vdash Q(b,x)\,[f_B\ b/x]\\
&\Delta_A, \Delta_B \vdash x = S_1\ w\,[f_A\ a/w]\,[f_B\ b/x]
\end{aligned}$$

are, as yet, unproven formulas, and

$$\begin{aligned}
\Delta_A\ &=_{df}\ [a\colon A]\\
\Delta_B\ &=_{df}\ [b\colon B,\ h\colon b = R_1\ a\,]\ .
\end{aligned}$$

Interestingly, this sequence of type inferences is invariant of the attributes of A, B, W and X, the predicates P and Q, and the functions f_A and f_B. In fact, it could be regarded as a template for constructing the proofs of *all* two-tier transformations of this kind, for it comprises

- a placeholder for a proof of the transformation between the attributes of one pair of classes, i.e.
$$\Delta_A \vdash P(a,w)\,[f_A\ a/w]\ ;$$
- a placeholder for a proof of the transformation between the attributes of the other pair of classes, i.e.
$$\Delta_B \vdash Q(b,x)\,[f_B\ b/x]\ ;$$
- a placeholder for a proof that one transformation nestles within the other, e.g.
$$\Delta_A, \Delta_B \vdash x = S_1\ w\,[f_A\ a/w]\,[f_B\ b/x]\ ;$$
- the necessary "glue" to bind the three placeholders together, in the form of an invariant sequence of type inferences.

If the first two unproven formulas are considered to be *horizontal* components, and the third unproven formula is considered to be a *vertical* component, the proof of the whole specification could be cast simply as the sum of the proofs of its horizontal and vertical components. This theme will crop again in the chapter on certified ordered model transformations. The proofs of the unproven formulas are as follows.

1.
$$\dfrac{\dfrac{\dfrac{\Delta_A \vdash A_1\, a\colon Nat}{\Delta_A \vdash A_1\, a = A_1\, a}\,(II)}{\Delta_A \vdash A_1\, a = W_1\, w\,[f_A\, a/w]}\,(=_{exp})}{\Delta_A \vdash P(a,w)\,[f_A\, a/w]}\,(=_{df})\,,$$

where

$$\begin{aligned} f_A &=_{df} \lambda a\,.\,@_W\,(A_1\, a)\,(f_B\,(R_1\, a)) \\ f_B &=_{df} \lambda b\,.\,(B_1\, b)\,. \end{aligned}$$

2.
$$\dfrac{\dfrac{\dfrac{\Delta_B \vdash B_1\, b\colon Nat}{\Delta_B \vdash B_1\, b = B_1\, b}\,(II)}{\Delta_B \vdash B_1\, b = X_1\, x\,[f_B\, b/x]}\,(=_{exp})}{\Delta_B \vdash Q(b,x)\,[f_B\, b/x]}\,(=_{df})\,.$$

3.
$$\dfrac{\dfrac{\dfrac{\Delta_A, \Delta_B \vdash f_B\,(R_1\, a)\colon X}{\Delta_A, \Delta_B \vdash f_B\,(R_1\, a) = f_B\,(R_1\, a)}\,(II)}{\Delta_A, \Delta_B \vdash f_B\,(R_1\, a) = S_1\,(f_A\, a)}\,(=_{exp}) \qquad \Delta_A, \Delta_B \vdash b = R_1\, a_{(Ass)}}{\Delta_A, \Delta_B \vdash f_B\, b = S_1\,(f_A\, a)}\,(IE)$$
$$\dfrac{}{\Delta_A, \Delta_B \vdash x = S_1\, w\,[f_A\, a/w]\,[f_B\, b/x]}\,(=_{exp})\,.$$

In summary, the function that implements the transformation between a source model and a target model is always the function that takes the root class of the source model to the root class of the target model (in this example, the function is f_A), where the root class is the one that directly or indirectly contains every other class in the model. One could imagine a transformation between two quite deeply nested containment models (A contains B, B contains C, C contains D, and so on) where the notion of a root class would have more resonance than it does here.

Coq Listing 32. *(Example B)* Coq supports more than one kind of existential quantification. The one that appears everywhere else in this thesis, namely `exists`, is of type `Prop`. However, the one below, which has a set-like syntax, is of type `Set`. The two are logically equivalent, but unlike the former, the latter is amenable to program extraction. In the following listing, a Haskell module `Main` containing a function `t` is extracted from a certified program `T`, by stripping away all its proof terms. What is left is a pure implementation function, which is smaller and leaner than `T` and therefore bound to run faster.

```
Record B : Set := {B1 : nat}.
Record A : Set := {A1 : nat; R1 : B}.
Record X : Set := {X1 : nat}.
Record W : Set := {W1 : nat; S1 : X}.

Definition P (a : A) (w : W) : Prop :=
```

```
   A1 a = W1 w.

Definition Q (b : B) (x : X) : Prop :=
   B1 b = X1 x.

Definition fB (b : B) : X :=
   Build_X (B1 b).

Definition fA (a : A) : W :=
   Build_W (A1 a) (fB (R1 a)).

Theorem T : forall a : A, { w : W | P a w /\
   forall b : B, b = R1 a -> exists x : X, Q b x /\ x = S1 w }.
Proof.
   intro a.
   ...
   reflexivity.
Qed.

Extraction Language Haskell.
Recursive Extraction T.

module Main where
...
fB :: B -> X
fB b =
  b1 b

fA :: A -> W
fA a =
  Build_W (a1 a) (fB (r1 a))

t :: A -> W
t a =
  fA a
```

4.4 Example C

This last example was motivated by the desire to formalise part of the behaviour of a commercial model to text generator [131], which distributes platform independent models across multiple processes. The class model is based on a fragment of the model of executable UML, as shown in Figure 4.4 (left), in which a class has operations, an operation has statements, and statements are either invocation statements or—for the purposes of this example—statements of other unspecified kinds. In this paradigm, a class is assigned to an operating system process during system configuration, so that its operational code will run in the context of that process.

Let i be an invocation statement in an operation of a class that is assigned to process

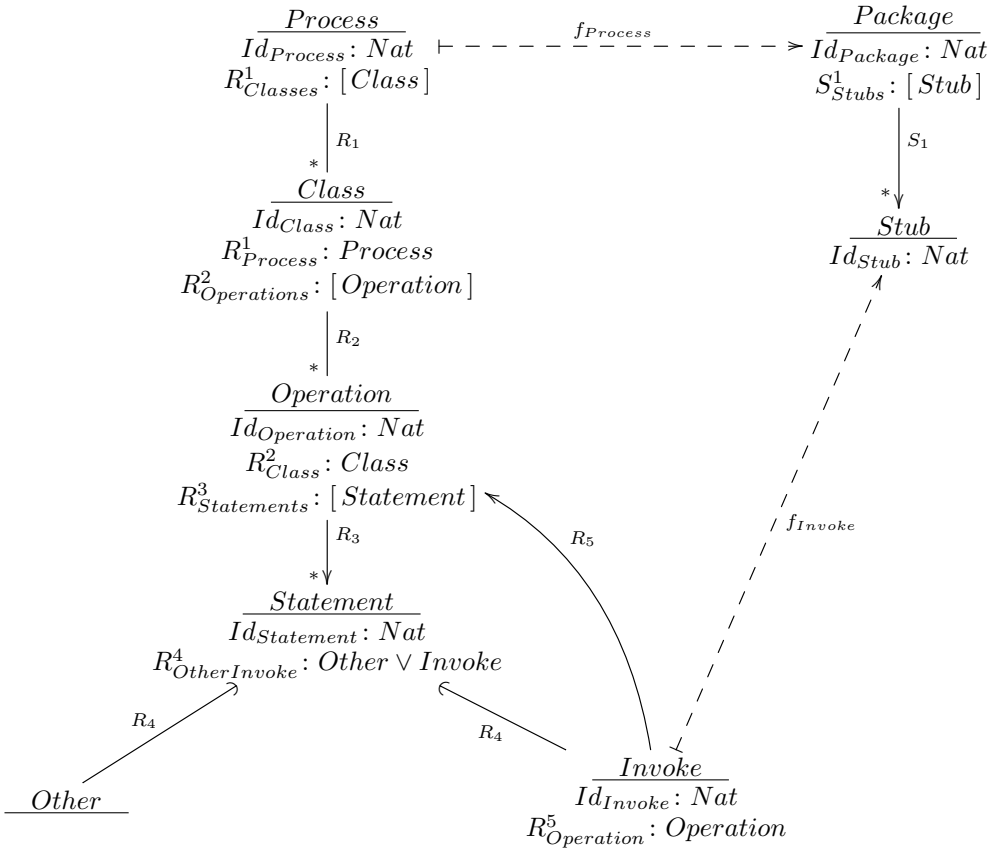

Figure 4.4: The source and target models for the third worked example.

p_i, and suppose that i invokes an operation in a class that is assigned to process p_j. If $p_i \neq p_j$, the invocation is said to be *remote*, otherwise it is *local*. Usually, a remote invocation statement is implemented by a *stub* in the source process and a *skeleton* in the destination process, to manage the flow of control and data between the invoking and invoked operations. An obvious question to ask is, "what stubs and skeletons are required to support the implementation of a particular source model?"

The specification of the transformation between the source and target models is given by U, where

$$
\begin{aligned}
U &=_{df} \forall p\colon Process\,.\,\exists a\colon Package\,.\,Id_{Process}\,p = Id_{Package}\,a \wedge \forall c\colon Class\,.\,V \\
V &=_{df} \quad c \in R^1_{Classes}\,p \rightarrow \forall o\colon Operation\,.\,W \\
W &=_{df} \quad\quad o \in R^2_{Operations}\,c \rightarrow \forall s\colon Statement\,.\,X \\
X &=_{df} \quad\quad\quad s \in R^3_{Statements}\,o \rightarrow Y \\
Y &=_{df} \quad\quad\quad\quad @^{-1}_{Other \vee Invoke}\,(\lambda d\,.\,U_0)\,(R^4_{OtherInvoke}\,s)\,(\lambda h\,.\,\top)\,(\lambda i\,.\,Z) \\
Z &=_{df} \quad\quad\quad\quad\quad p \neq R^1_{Process}\,(R^2_{Class}\,(R^5_{Operation}\,i)) \rightarrow \\
&\quad\quad\quad\quad\quad\quad \exists t\colon Stub\,.\,Id_{Operation}\,(R^5_{Operation}\,i) = Id_{Stub}\,t \wedge t \in S^1_{Stubs}\,a\ ,
\end{aligned}
$$

i.e. each process p is transformed into a package a, where a package is a collection of stubs, and each remote invocation statement in p is transformed into a stub in a.[2] The inequality operator in formula Z is defined by the specific equality function $Equal_{Process}$ below, where an equality function in general is defined as follows.

Definition 32. *(Equality Function)* An equality function over type A is a term $Equal_A$ of type $A \to A \to Bool$, such that

$$\forall a_1 \colon A \,.\, \forall a_2 \colon A \,.\, a_1 =_A a_2 \to Equal_A\, a_1\, a_2 =_{Bool} True$$
$$\forall a_1 \colon A \,.\, \forall a_2 \colon A \,.\, a_1 \neq_A a_2 \to Equal_A\, a_1\, a_2 =_{Bool} False\ .$$

Hence, if two objects are equal, the equality function returns $True$.

Definition 33. *($Equal_{Process}$)* The equality function over type $Process$ is given by

$$Equal_{Process} =_{df} \lambda p_1 \,.\, \lambda p_2 \,.\, (\mathit{if}\ Id_{Process}\, p_1 =_{Nat} Id_{Process}\, p_2)\ \mathit{then}\ True\ \mathit{else}\ False\ .$$

4.4.1 Substitutions

The proof of U below, in Section 4.4.3, makes use of the following substitutions. In step 19, ρ replaces package a with an existential witness $f_{Process}\, p$ for some process p. Further, in the premisses of the $([\,Class\,]\,E)$ rule in step 18, σ_c^0, σ_c^1 and σ_c^2 replace the specific list of classes $R^1_{Classes}\, p$ with the more general $Nil_{[\,Class\,]}$, l_c and $c' :: l_c$ respectively. Similarly for σ_o^0, σ_o^1 and σ_o^2 in step 14, and σ_s^0, σ_s^1 and σ_s^2 in step 10. Finally, in step 4, ρ_t replaces stub t with an existential witness $@_{Stub}\,(Id_{Operation}\,(R^5_{Operation}\, i))$ for some invocation statement i.

$$\begin{aligned}
\rho &=_{df} [f_{Process}\, p/a] \\
\sigma_c^0 &=_{df} [Nil_{[\,Class\,]}/R^1_{Classes}\, p] \\
\sigma_c^1 &=_{df} [l_c/R^1_{Classes}\, p] \\
\sigma_c^2 &=_{df} [c' :: l_c/R^1_{Classes}\, p] \\
\sigma_o^0 &=_{df} [Nil_{[\,Operation\,]}/R^2_{Operations}\, c'] \\
\sigma_o^1 &=_{df} [l_o/R^2_{Operations}\, c'] \\
\sigma_o^2 &=_{df} [o' :: l_o/R^2_{Operations}\, c'] \\
\sigma_s^0 &=_{df} [Nil_{[\,Statement\,]}/R^3_{Statements}\, o'] \\
\sigma_s^1 &=_{df} [l_s/R^3_{Statements}\, o'] \\
\sigma_s^2 &=_{df} [s' :: l_s/R^3_{Statements}\, o'] \\
\rho_t &=_{df} [@_{Stub}\,(Id_{Operation}\,(R^5_{Operation}\, i))/t]\ ,
\end{aligned}$$

where

$$f_{Process} =_{df} \lambda p \,.\, @_{Package}\,(Id_{Process}\, p)$$

[2] The construction of skeletons is ommited for pedagogical reasons.

$$(f_{Invokes}\, p\, (R^4_{Invokes}\, (R^3_{Statements'}\, (R^2_{Operations'}\, (R^1_{Classes}\, p)))))$$

and

$$\begin{aligned} f_{Invokes} \;=_{df}\; & \lambda p\,.\,\lambda l\,.\,@^{-1}_{[\,Invoke\,]}\,(\lambda l\,.\,[\,Stub\,])\; l\; Nil_{[\,Stub\,]} \\ & (\lambda i\,.\,\lambda l'\,.\,\lambda h\,.\,if\;(p = R^1_{Process}\,(R^2_{Class}\,(R^5_{Operation}\,i))) \\ & \quad then\;(f_{Invokes}\, p\, l')\; else\;(f_{Invoke}\, i) :: f_{Invokes}\, p\, l') \end{aligned}$$

and

$$f_{Invoke} \;=_{df}\; \lambda i\,.\,@_{Stub}\,(Id_{Operation}\,(R^5_{Operation}\,i))$$

and

$$\begin{aligned} R^4_{Invokes} \;=_{df}\; & \lambda l\,.\,@^{-1}_{[\,Statement\,]}\,(\lambda l\,.\,[\,Invoke\,])\; l\; Nil_{[\,Invoke\,]} \\ & (\lambda s\,.\,\lambda l'\,.\,@^{-1}_{Other \vee Invoke}\,(\lambda l\,.\,[\,Invoke\,])\; s\; (\lambda h\,.\,R^4_{Invokes}\, l')\; (\lambda i\,.\,i :: R^4_{Invokes}\, l')) \end{aligned}$$

and

$$\begin{aligned} R^3_{Statements'} \;=_{df}\; & \lambda l\,.\,@^{-1}_{[\,Operation\,]}\,(\lambda l\,.\,[\,Statement\,])\; l\; Nil_{[\,Statement\,]} \\ & (\lambda o\,.\,\lambda l'\,.\,R^3_{Statements}\, o +\!\!+ R^3_{Statements'}\, l') \end{aligned}$$

and $+\!\!+$ is the list append operator. $R^2_{Operations'}$ is defined in a similar manner to $R^3_{Statements'}$. Moreover, $R^1_{Classes}$, $R^1_{Process}$, $R^2_{Operations}$, R^2_{Class}, $R^3_{Statements}$, $R^4_{OtherInvoke}$ and $R^5_{Operation}$ are the usual relation functions.

4.4.2 Contexts

The proof of U makes use of the following contexts.

$$\begin{aligned} \Delta_p \;&=_{df}\; [p\colon Process] \\ \Delta_c \;&=_{df}\; \Delta_p, [c\colon Class] \\ \Delta'_c \;&=_{df}\; \Delta_c, [c'\colon Class,\, l_c\colon [\,Class\,],\, ih_c\colon V\,\rho\,\sigma^1_c\,] \\ {}^{=}\Delta_o \;&=_{df}\; \Delta'_c, [h_c\colon c = c',\, o\colon Operation\,] \\ {}^{\in}\Delta_o \;&=_{df}\; \Delta'_c, [h_c\colon c \in l_c,\, o\colon Operation\,] \\ {}^{=}\Delta'_o \;&=_{df}\; {}^{=}\Delta_o, [o'\colon Operation,\, l_o\colon [\,Operation\,],\, ih_o\colon W\,\rho\,\sigma^2_c\,[c'/c]\,\sigma^1_o\,] \\ {}^{=}\Delta_s \;&=_{df}\; \Delta'_o, [h_o\colon o = o',\, s\colon Statement\,] \\ {}^{=}\Delta'_s \;&=_{df}\; {}^{=}\Delta_s, [s'\colon Statement,\, l_s\colon [\,Statement\,],\, ih_s\colon X\,\rho\,\sigma^2_c\,[c'/c]\,\sigma^2_o\,[o'/o]\,\sigma^1_s\,] \\ {}^{=}\Delta \;&=_{df}\; {}^{=}\Delta'_s, [h_s\colon s' = s] \\ {}^{=}\Delta' \;&=_{df}\; {}^{=}\Delta, [i\colon Invoke,\, h_e\colon p \neq R^1_{Process}\,(R^2_{Class}\,(R^5_{Operation}\,i))\,] \\ {}^{\in}\Delta \;&=_{df}\; {}^{=}\Delta'_s, [h_s\colon s \in l_s] \\ {}^{\in}\Delta' \;&=_{df}\; {}^{\in}\Delta, [i\colon Invoke,\, h_e\colon p \neq R^1_{Process}\,(R^2_{Class}\,(R^5_{Operation}\,i))\,]\,. \end{aligned}$$

4.4.3 Proof

The proof is composed of 19 steps. The reader is advised to start with step 19 and work backwards.

1.
$$\dfrac{\dfrac{\dfrac{}{{}^{=}\Delta, [h\colon Other] \vdash \top}\,(\top I)}{{}^{=}\Delta \vdash \forall h\colon Other\,.\,\top}\,(\forall I)}{{}^{=}\Delta \vdash \forall h\colon Other\,.\,Y\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2\;[s'/s]\;[inl\;h/R^4_{OtherClass}\;s']}\,(=_{exp})\,.$$

2.
$$\dfrac{\dfrac{\dfrac{{}^{=}\Delta' \vdash Id_{Operation}\,(R^5_{Operation}\,i)\colon Nat}{{}^{=}\Delta' \vdash Id_{Operation}\,(R^5_{Operation}\,i) = Id_{Operation}\,(R^5_{Operation}\,i)}\,(II)}{{}^{=}\Delta' \vdash Id_{Operation}\,(R^5_{Operation}\,i) = Id_{Stub}\,(@_{Stub}\,(Id_{Operation}\,(R^5_{Operation}\,i)))}\,(=_{exp})}{{}^{=}\Delta' \vdash (Id_{Operation}\,(R^5_{Operation}\,i) = Id_{Stub}\,t)\;\rho_t}\,(=_{df})\,.$$

3.
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{{}^{=}\Delta' \vdash f_{Invoke}\,i\colon Stub}{{}^{=}\Delta' \vdash f_{Invoke}\,i =_{Stub} f_{Invoke}\,i}\,(II)}{{}^{=}\Delta' \vdash t\;\rho_t = f_{Invoke}\,i}\,(=_{exp})}{{}^{=}\Delta' \vdash t\;\rho_t \in f_{Invoke}\,i :: f_{Invokes}\,p\,l_i}\,(\vee I_1)}{{}^{=}\Delta' \vdash t\;\rho_t \in if\;False\;then\;(f_{Invokes}\,p\,l_i)\;else\;f_{Invoke}\,i :: f_{Invokes}\,p\,l_i}\,(=_{exp}) \qquad {}^{=}\Delta' \vdash (p = p_i) = False_{(Ass)}}{{}^{=}\Delta' \vdash t\;\rho_t \in if\;p = p_i\;then\;(f_{Invokes}\,p\,l_i)\;else\;(f_{Invoke}\,i :: f_{Invokes}\,p\,l_i)}\,(IE)}{{}^{=}\Delta' \vdash t\;\rho_t \in f_{Invokes}\,p\,l_i}\,(=_{exp})}{{}^{=}\Delta' \vdash (t \in S^1_{Stubs}\,a)\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2\;[s'/s]\;\rho_t}\,(=_{exp})\,,$$

where

$$\begin{aligned} l_i &=_{df} R^4_{Invokes}\,(l_s +\!\!+ R^3_{Statements'}\,l_o +\!\!+ (R^2_{Operations'}\,l_c)) \\ p_i &=_{df} R^1_{Process}\,(R^2_{Class}\,(R^5_{Operation}\,i))\,. \end{aligned}$$

4. Use steps 2 and 3.
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{{}^{=}\Delta' \vdash (Id_{Operation}\,(R^5_{Operation}\,i) = Id_{Stub}\,t)\;\rho_t \quad {}^{=}\Delta' \vdash (t \in S^1_{Stubs}\,a)\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2\;[s'/s]\;\rho_t}{{}^{=}\Delta' \vdash (Id_{Operation}\,(R^5_{Operation}\,i) = Id_{Stub}\,t \wedge t \in S^1_{Stubs}\,a)\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2\;[s'/s]\;\rho_t}\,(\wedge I)}{{}^{=}\Delta' \vdash (\exists t\colon Stub\,.\,Id_{Operation}\,(R^5_{Operation}\,i) = Id_{Stub}\,t \wedge t \in S^1_{Stubs}\,a)\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2\;[s'/s]}\,(\exists I)}{{}^{=}\Delta \vdash \forall i\colon Invoke\,.\,Z\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2\;[s'/s]}\,(\to I)\,(\forall I)}{{}^{=}\Delta \vdash \forall i\colon Invoke\,.\,Y\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2\;[s'/s]\;[inr\;i/R^4_{OtherClass}\;s']}\,(=_{exp})\,.$$

5. Use steps 1 and 4.

$$\dfrac{\dfrac{\dfrac{{}^{=}\Delta \vdash \forall h\colon Other\,.\,Y\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2\ [o'/o]\ \sigma_s^2\ [s'/s]\ [inl\ h/R^4_{OtherClass}\ s'] \quad {}^{=}\Delta \vdash \forall i\colon Invoke\,.\,Y\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2\ [o'/o]\ \sigma_s^2\ [s'/s]\ [inr\ i/R^4_{OtherClass}\ s']}{{}^{=}\Delta \vdash Y\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2\ [o'/o]\ \sigma_s^2\ [s'/s]}\ (\vee E) \quad {}^{=}\Delta \vdash s = s'_{(Ass)}}{{}^{=}\Delta \vdash Y\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2\ [o'/o]\ \sigma_s^2}\ (IE)}{{}^{=}\Delta'_s \vdash s = s' \to Y\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2\ [o'/o]\ \sigma_s^2}\ (\forall I)(\to I)\ .$$

6. The proof of

$$^{\in}\Delta \vdash \forall h\colon Other\,.\,Y\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2\ [o'/o]\ \sigma_s^2\ [inl\ h/R^4_{OtherClass}\ s]$$

is similar to step 1.

7. To add variety, here is a Coq proof of

$$^{\in}\Delta \vdash \forall i\colon Invoke\,.\,Y\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2\ [o'/o]\ \sigma_s^2\ [inr\ i/R^4_{OtherClass}\ s]\ .$$

Coq Listing 33. *(Step 7)* In the listing below, the context above the dashed line is more or less $^{\in}\Delta$, and the goal below the line (which is one of three outstanding goals, the others being associated with steps 12 and 16) is the normalised form of

$$Y\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2\ [o'/o]\ \sigma_s^2\ [inr\ i/R^4_{OtherClass}\ s]\ ,$$

i.e. the body of the universal quantification. Note that i is already in the context (Coq automatically put it there). The proof below decomposes the first goal into smaller and smaller formulas until a point is reached where each one can be trivially proved from the context, usually by means of the `assumption` tactic.

```
p : Process
c : Class
c' : Class
lc : list Class
ihc : In c lc -> ...
hc : c' = c
o : Operation
o' : Operation
lo : list Operation
iho : In o lo -> ...
ho : o' = o
s : Statement
s' : Statement
ls : list Statement
i : Invoke
ihs : In s ls -> ...
hs : Equal_Process p (R1_Process (R2_Class (R5_Operation i))) = false -> ...
```

```
______________________________________(1/3)
Equal_Process p (R1_Process (R2_Class (R5_Operation i))) = false ->
exists t : Stub,
  Id_Stub t = Id_Operation (R5_Operation i) /\
  In t
    (f_Invokes p
       match R4_Other_Invoke s' with
       | inl _ => R4_Invokes (ls ++ R3_Statements' (lo ++ R2_Operations' lc))
       | inr i0 =>
           i0 :: R4_Invokes (ls ++ R3_Statements' (lo ++ R2_Operations' lc))
       end)
Proof.
   intro he.
   apply hs in he.
   destruct he as (wt , he).
   exists (wt).
   destruct he as [he hf].
   split.
   assumption.
   case (R4_Other_Invoke s').
   intro.
   assumption.
   intro i'.
   simpl.
   case (Equal_Process p (R1_Process (R2_Class (R5_Operation i')))).
   assumption.
   right.
   assumption.
```

8. Use steps 6 and 7.

$$\dfrac{\dfrac{\begin{array}{c}{}^{\in}\Delta \vdash \forall h\colon Other\,.\,Y\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2\;[inl\;h/R^4_{OtherClass}\;s] \\ {}^{\in}\Delta \vdash \forall i\colon Invoke\,.\,Y\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2\;[inr\;i/R^4_{OtherClass}\;s]\end{array}}{{}^{\in}\Delta \vdash Y\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2}\;(\vee E)}{{}^{=}\Delta'_s \vdash s \in l_s \to Y\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2}\;(\forall I)(\to I)\;.$$

9. The proof of

$$ {}^{=}\Delta_s \vdash X\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^0 $$

is similar to step 17.

10. Use steps 5, 8 and 9.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}{}^{=}\Delta'_s \vdash s = s' \to Y\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2 \\ {}^{=}\Delta'_s \vdash s \in l_s \to Y\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2\end{array}}{{}^{=}\Delta'_s \vdash s \in s' :: l_s \to Y\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2}\;(\vee E)}{\begin{array}{l}{}^{=}\Delta_s \vdash \forall o'\,.\,\forall l_o\,.\,X\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^1 \to X\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^2 \\ {}^{=}\Delta_s \vdash X\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]\;\sigma_s^0\end{array}}\;(\to I)\,(\forall I)_*}{{}^{=}\Delta_s \vdash X\;\rho\;\sigma_c^2\;[c'/c]\;\sigma_o^2\;[o'/o]}\;([\,Statement\,]\,E)\;.$$

11. Use step 10.

$$\dfrac{\dfrac{{}^{=}\Delta_s \vdash o = o'_{(Ass)} \quad {}^{=}\Delta_s \vdash X\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2\ [o/o']}{{}^{=}\Delta_s \vdash X\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2}\ (IE)}{{}^{=}\Delta'_o \vdash o = o' \to \forall s\colon Statement\,.\,X\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2}\ (\forall I)(\to I)\ .$$

12. The proof of

$${}^{=}\Delta'_o \vdash o \in l_o \to \forall s\colon Statement\,.\,X\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2$$

is similar to step 8.

13. The proof of

$${}^{=}\Delta_o \vdash W\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^0$$

is similar to step 17.

14. Use steps 11, 12 and 13.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}{}^{=}\Delta'_o \vdash o = o' \to \forall s\colon Statement\,.\,X\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2 \\ {}^{=}\Delta'_o \vdash o \in l_o \to \forall s\colon Statement\,.\,X\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2\end{array}}{{}^{=}\Delta'_o \vdash o \in o' :: l_o \to \forall s\colon Statement\,.\,X\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2}\ (\vee E)}{{}^{=}\Delta_o \vdash \forall o'\,.\,\forall l_o\,.\,W\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^1 \to W\ \rho\ \sigma_c^2\ [c'/c]\ \sigma_o^2}\ (\to I)\,(\forall I)_*}{{}^{=}\Delta_o \vdash W\ \rho\ \sigma_c^2\ [c'/c]}\ ([\,Operation\,]\,E)\ .$$

15. Use step 14.

$$\dfrac{\dfrac{{}^{=}\Delta_o \vdash c = c'_{(Ass)} \quad {}^{=}\Delta_o \vdash W\ \rho\ \sigma_c^2\ [c'/c]}{{}^{=}\Delta_o \vdash W\ \rho\ \sigma_c^2}\ (IE)}{\Delta'_c \vdash c = c' \to \forall o\colon Operation\,.\,W\ \rho\ \sigma_c^2}\ (\forall I)(\to I)\ .$$

16. The proof of

$$\Delta'_c \vdash c \in l_c \to \forall o\colon Operation\,.\,W\,\rho\,\sigma_c^2$$

is similar to step 8.

17.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta_c, [h_c\colon c \in Nil_{[Class]}] \vdash c \in Nil_{[Class]}{}_{(Ass)}}{\Delta_c, [h_c\colon c \in Nil_{[Class]}] \vdash \bot}\ (=_{df})}{\Delta_c, [h_c\colon c \in Nil_{[Class]}] \vdash \forall o\colon Operation\,.\,W\ \rho\ \sigma_c^0}\ (\bot E)}{\Delta_c \vdash V\,\rho\,\sigma_c^0}\ (\to I)\ .$$

18. Use steps 15, 16 and 17.

$$\dfrac{\Delta_c \vdash V\,\rho\,\sigma_c^0 \qquad \dfrac{\dfrac{\begin{array}{c}\Delta'_c \vdash c = c' \to \forall o\colon Operation\,.\,W\,\rho\,\sigma_c^2 \\ \Delta'_c \vdash c \in l_c \to \forall o\colon Operation\,.\,W\,\rho\,\sigma_c^2\end{array}}{\Delta'_c \vdash c \in c' :: l_c \to \forall o\colon Operation\,.\,W\,\rho\,\sigma_c^2}\ (\vee E)}{\Delta_c \vdash \forall c'\,.\,\forall l_c\,.\,V\,\rho\,\sigma_c^1 \to V\,\rho\,\sigma_c^2}\ (\to I)\,(\forall I)_*}{\Delta_c \vdash V\,\rho}\ ([\,Class\,]\,E)\ .$$

19. Use step 18.

$$\dfrac{\dfrac{\dfrac{\Delta_p \vdash (Id_{Process}\, p = Id_{Package}\, a)\, \rho \quad \dfrac{\Delta_c \vdash V\, \rho}{\Delta_p \vdash \forall c\colon Class\,.\,V\, \rho}\,(\forall I)}{\Delta_p \vdash (Id_{Process}\, p = Id_{Package}\, a\ \wedge \forall c\colon Class\,.\,V)\, \rho}\,(\wedge I)}{\Delta_p \vdash \exists a\colon Package\,.\,Id_{Process}\, p = Id_{Package}\, a\ \wedge \forall c\colon Class\,.\,V}\,(\exists I)}{\vdash U}\,(\forall I)\ .$$

4.4.4 Conclusions

The proof shows that U is inhabited by

$$\lambda p\,.\,\langle f_{Process}\, p,\, q\rangle\ ,$$

where q is a proof of the postcondition of the transformation in the context of Δ_p, i.e.

$$\Delta_p \vdash q\colon (Id_{Process}\, p = Id_{Package}\, a\ \wedge \forall c\colon Class\,.\,V)\, \rho\ .$$

Coq Listing 34. *(Example C)* The source model is defined as a `CoInductive` type to allow its object models to be cyclic, and the cyclic object model in Figure 4.5 is instantiated as a `CoFixpoint`. When the inhabitant of U above (which is called `ProcessPackage` below) is computed on processes `p1` and `p2`, the target proofs—which are heavily elided below—show that `p1` requires two stubs (for operations `o4` and `o3`), and `p2` requires none.

```
CoInductive Process : Set :=
   Build_Process : nat -> list Class -> Process
with Class : Set :=
   Build_Class : nat -> Process -> list Operation -> Class
with Operation : Set :=
   Build_Operation : nat -> Class -> list Statement -> Operation
with Statement : Set :=
   Build_Statement : nat -> Other + Invoke -> Statement
with Other : Set :=
   Build_Other : Other
with Invoke : Set :=
   Build_Invoke : nat -> Operation -> Invoke.

CoFixpoint p1 : Process :=
   Build_Process 1 (c1 :: nil)
with p2 : Process :=
   Build_Process 2 (c2 :: nil)
with c1: Class :=
   Build_Class 1 p1 (o1 :: o2 :: nil)
with c2 : Class :=
   Build_Class 2 p2 (o3 :: o4 :: nil)
with o1 : Operation :=
   Build_Operation 1 c1 nil
with o2 : Operation :=
```

```
   Build_Operation 2 c1 (s1 :: s2 :: s3 :: nil)
with o3 : Operation :=
   Build_Operation 3 c2 nil
with o4 : Operation :=
   Build_Operation 4 c2 nil
with s1 : Statement :=
   Build_Statement 1 (inr Other i1)
with s2 : Statement :=
   Build_Statement 2 (inr Other i2)
with s3 : Statement :=
   Build_Statement 3 (inr Other i3)
with i1 : Invoke :=
   Build_Invoke 1 o1
with i2 : Invoke :=
   Build_Invoke 2 o4
with i3 : Invoke :=
   Build_Invoke 3 o3.

> Compute (ProcessPackage p1).
     = ex_intro
         (fun a : Package =>
(elided)
           end)) (Build_Package 1 (Build_Stub 4 :: Build_Stub 3 :: nil))
(elided)

> Compute (ProcessPackage p2).
     = ex_intro
         (fun a : Package =>
(elided)
           end)) (Build_Package 2 nil)
(elided)
```

4.5 Recursive Specifications

The challenge in specifying the logical interpretation of an ordered model transformation, as defined later in Section 5.3, was in finding a way to formalise the specification of an arbitrarily large set of nested universal quantifications, ranging over a set of dependent types whose inhabitants varied with the values of bound variables. This section describes, by means of a relatively simple example, how this challenge was met. Let

$$\begin{aligned} P(n) &=_{df} 0 \leq n \\ Q(n,m) &=_{df} 0 \leq n+m\ , \end{aligned}$$

where type $x \leq y$ is as defined in Section 4.1, and let

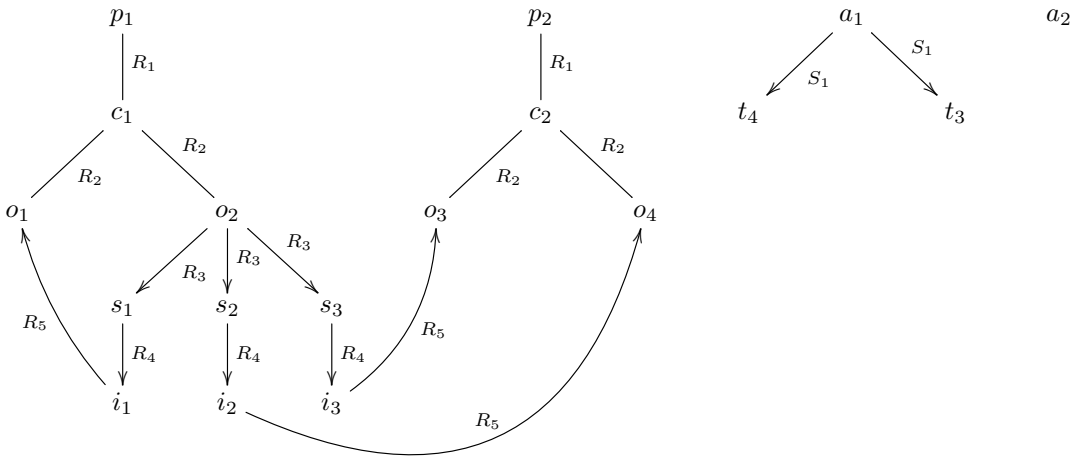

Figure 4.5: On the left, source objects from the `CoFixpoint` in Listing 34; on the right, target objects a_1 and a_2, where a_1 was derived from p_1, and a_2 was derived from p_2.

$$
\begin{aligned}
\tau_n \;=_{df}\; & \forall x\colon Nat\,.\,P(x) \wedge \\
& \quad \forall y_0\colon Nat\,.\,Q(x, y_0) \wedge \\
& \quad\quad \forall y_1\colon Nat\,.\,Q(y_0, y_1) \wedge \\
& \quad\quad\quad \forall y_2\colon Nat\,.\,Q(y_1, y_2) \wedge \\
& \quad\quad\quad\quad \ldots \\
& \quad\quad\quad\quad\quad \forall y_n\colon Nat\,.\,Q(y_{n-1}, y_n)\ ,
\end{aligned}
$$

for natural numbers n. Clearly, x is bound in P; and x and y_0, then y_0 and y_1, then y_1 and y_2 and so on, are bound in Q. Further, the universally bound variable at one level becomes the first argument of Q at the next level. Individually, every type τ_n can be shown to be inhabited, no matter what the value of n (e.g. Lemma 1 below proves that τ_0 is inhabited). However, the real challenge is in devising a scheme that is capable of formally specifying and proving τ_n for all values of n, in one fell swoop.

Lemma 1. $\vdash \forall x\colon Nat\,.\,P(x) \wedge \forall y\colon Nat\,.\,Q(x, y)$.

Proof. The proof uses objects L_1 and L_2, as defined in Section 4.5.3.

$$
\dfrac{\dfrac{\dfrac{\begin{array}{l} L_1\colon \forall x\colon Nat\,.\,P(x) \\ [x\colon Nat] \vdash x\colon Nat_{(Ass)} \end{array}}{[x\colon Nat] \vdash P(x)}\,(\forall E) \quad \dfrac{\begin{array}{l} L_2\colon \forall x\colon Nat\,.\,\forall y\colon Nat\,.\,Q(x, y) \\ [x\colon Nat] \vdash x\colon Nat_{(Ass)} \end{array}}{[x\colon Nat] \vdash \forall y\colon Nat\,.\,Q(x, y)}\,(\forall E)}{[x\colon Nat] \vdash P(x) \wedge \forall y\colon Nat\,.\,Q(x, y)}\,(\wedge I)}{\vdash \forall x\colon Nat\,.\,P(x) \wedge \forall y\colon Nat\,.\,Q(x, y)}\,(\forall I)\ .
$$

□

4.5.1 Specification

Since

$$\begin{aligned}
\tau_0 &=_{df} \forall x\colon Nat\,.\,P(x) \wedge \forall y_0\colon Nat\,.\,Q(x,y_0)\\
\tau_1 &=_{df} \forall x\colon Nat\,.\,P(x) \wedge \forall y_0\colon Nat\,.\,Q(x,y_0) \wedge \forall y_1\colon Nat\,.\,Q(y_0,y_1)\\
\tau_2 &=_{df} \forall x\colon Nat\,.\,P(x) \wedge \forall y_0\colon Nat\,.\,Q(x,y_0) \wedge \forall y_1\colon Nat\,.\,Q(y_0,y_1) \wedge \forall y_2\colon Nat\,.\,Q(y_1,y_2)\,,
\end{aligned}$$

it can be shown that

$$\tau_n =_{df} \forall x\colon Nat\,.\,P(x) \wedge \forall n\colon Nat\,.\,Spec\,n\,x\,,$$

where $Spec\,0\,x$ returns

$$\forall y_0\colon Nat\,.\,Q(x,y_0)\,,$$

$Spec\,1\,x$ returns

$$\forall y_0\colon Nat\,.\,Q(x,y_0) \wedge \forall y_1\colon Nat\,.\,Q(y_0,y_1)\,,$$

and so on. Clearly, $Spec$ is a recursive function over Nat.

Definition 34. *(Spec)* A nested set of $n+1$ universal quantifications over Nat, rooted at x and ranging over Q, is given by $Spec\,n\,x$, where

$$\begin{aligned}
Spec &=_{df} \lambda n\,.\,\lambda x\,.\,@^{-1}_{Nat}\,T\,n\,i\,j\\
T &=_{df} \lambda n\,.\,U_0\\
i &=_{df} \forall y\colon Nat\,.\,Q(x,y)\\
j &=_{df} \lambda n\,.\,\lambda h\,.\,\forall y\colon Nat\,.\,Q(x,y) \wedge Spec\,n\,y\,.
\end{aligned}$$

Note that i is allied to the base case, and j is allied to the recursive step; and further that x is free in i and j so that its value can be passed in from outside (hence the many substitutions in Example 11).

Example 11. Compute $Spec\,2\,\alpha$.

$$\begin{aligned}
Spec\,2\,\alpha &\twoheadrightarrow @^{-1}_{Nat}\,P\,2\,(i\,[\alpha/x])\,(j\,[\alpha/x])\\
&\twoheadrightarrow (j\,[\alpha/x])\,1\,(@^{-1}_{Nat}\,P\,1\,(i\,[\alpha/x])\,(j\,[\alpha/x]))\\
&\twoheadrightarrow \forall y\colon Nat\,.\,Q(\alpha,y) \wedge Spec\,1\,y\\
&\twoheadrightarrow \forall y\colon Nat\,.\,Q(\alpha,y) \wedge @^{-1}_{Nat}\,P\,1\,(i\,[y/x])\,(j\,[y/x])\\
&\twoheadrightarrow \forall y\colon Nat\,.\,Q(\alpha,y) \wedge (j\,[y/x])\,0\,(@^{-1}_{Nat}\,P\,0\,(i\,[y/x])\,(j\,[y/x]))\\
&\twoheadrightarrow \forall y\colon Nat\,.\,Q(\alpha,y) \wedge \forall y'\colon Nat\,.\,Q(y,y') \wedge Spec\,0\,y'\\
&\twoheadrightarrow \forall y\colon Nat\,.\,Q(\alpha,y) \wedge \forall y'\colon Nat\,.\,Q(y,y') \wedge \forall y''\colon Nat\,.\,Q(y',y'')\\
&\rightarrow_{\alpha} \forall y_0\colon Nat\,.\,Q(\alpha,y_0) \wedge \forall y_1\colon Nat\,.\,Q(y_0,y_1) \wedge \forall y_2\colon Nat\,.\,Q(y_1,y_2)\,.
\end{aligned}$$

Coq Listing 35. *(Spec)* In Coq, `Spec` is defined by a `Fixpoint`. The `delta` evaluation of `Spec 2 a`, suitably elided, produces an object of type `Prop`, i.e. a proposition, which is essentially the same as the hand rolled one above.

```
Definition P (n : nat) := 0 <= n.
Definition Q (n m : nat) := 0 <= n + m.

Fixpoint Spec (n : nat) (x : nat) {struct n}  :=
   match n with |
      0 => forall y : nat, Q x y |
      S n' => forall y : nat, Q x y /\ Spec n' y
   end.

Variable a : nat.
Eval cbv delta in (Spec 2 a).
     = forall y : nat, 0 <= (fix plus ...) a y /\
       (forall y0 : nat, 0 <= (fix plus ...) y y0 /\
        (forall y1 : nat, 0 <= (fix plus ...) y0 y1))
     : Prop
```

4.5.2 Proof

The section defines a complementary function, $Proof\ n\ x$, which proves $Spec\ n\ x$ for all n.

Definition 35. *(Proof)* The proof of $Spec\ n\ x$ is given by $Proof\ n\ x$, where

$$\begin{aligned} Proof &=_{df} \lambda n\,.\,\lambda x\,.\,@^{-1}_{Nat}\ P\ n\ i\ j \\ P &=_{df} \lambda n\,.\,Spec\ n\ x \\ i &=_{df} \lambda y\,.\,L_2\ x\ y \\ j &=_{df} \lambda n\,.\,\lambda h\,.\,\lambda y\,.\,\langle L_2\ x\ y,\ Proof\ n\ y\rangle\ , \end{aligned}$$

and L_2 is defined in Section 4.5.3.

Example 12. Compute $Proof\ 2\ \alpha$.

$$\begin{aligned} Proof\ 2\ \alpha &\twoheadrightarrow @^{-1}_{Nat}\ (P\ [\alpha/x])\ 2\ (i\ [\alpha/x])\ (j\ [\alpha/x]) \\ &\twoheadrightarrow (j\ [\alpha/x])\ 1\ (@^{-1}_{Nat}\ (P\ [\alpha/x])\ 1\ (i\ [\alpha/x])\ (j\ [\alpha/x])) \\ &\twoheadrightarrow \lambda y\,.\,\langle L_2\ \alpha\ y,\ Proof\ 1\ y\rangle \\ &\twoheadrightarrow \lambda y\,.\,\langle L_2\ \alpha\ y,\ (@^{-1}_{Nat}\ (P\ [y/x])\ 1\ (i\ [y/x])\ (j\ [y/x]))\rangle \\ &\twoheadrightarrow \lambda y\,.\,\langle L_2\ \alpha\ y,\ ((j\ [y/x])\ 1\ (@^{-1}_{Nat}\ (P\ [y/x])\ 0\ (i\ [y/x])\ (j\ [y/x])))\rangle \\ &\twoheadrightarrow \lambda y\,.\,\langle L_2\ \alpha\ y,\ (\lambda y'\,.\,\langle L_2\ y\ y',\ Proof\ 0\ y'\rangle)\rangle \\ &\twoheadrightarrow \lambda y\,.\,\langle L_2\ \alpha\ y,\ (\lambda y'\,.\,\langle L_2\ y\ y',\ (@^{-1}_{Nat}\ (P\ [y'/x])\ 0\ (i\ [y'/x])\ (j\ [y'/x]))\rangle)\rangle \\ &\twoheadrightarrow \lambda y\,.\,\langle L_2\ \alpha\ y,\ (\lambda y'\,.\,\langle L_2\ y\ y',\ i\ [y'/x]\rangle)\rangle \\ &\twoheadrightarrow \lambda y\,.\,\langle L_2\ \alpha\ y,\ (\lambda y'\,.\,\langle L_2\ y\ y',\ \lambda y''\,.\,L_2\ y'\ y''\rangle)\rangle\ . \end{aligned}$$

Finally, to complete the circle:

Lemma 2. $Proof\ 2\ \alpha$ is an object of $Spec\ 2\ \alpha$.

Proof. In Coq.

Coq Listing 36. *(Proof 2 a)* The proof is hardly surprising given that `Proof n x` is *defined* to be an object of type `Spec n x`.

```
Fixpoint Proof (n : nat) (x : nat)  {struct n} : Spec n x :=
   match n as n0 return (Spec n0 x) with |
      0 => fun y : nat => L2 x y |
      S n' => fun y : nat => conj (L2 x y) (Proof n' y)
   end.

Goal Spec 2 a.
Proof.
   exact (Proof 2 a).
Qed.
```

□

4.5.3 Additional Objects (L_1, L_2)

This section defines objects L_1 and L_2. These objects are referenced by Lemma 1 and the definition of $Proof$.

Lemma 3. $\vdash \forall x\colon Nat\,.\,P(x)$.

Proof. By induction on x. If $L_1\colon \forall x\colon Nat\,.\,P(x)$, then

$$L_1 =_{df} \lambda x\,.\,@^{-1}_{Nat}\,(\lambda n\,.\,0 \leq n)\,x\,(@^1_{\leq}\,0)\,(\lambda n\,.\,\lambda h\,.\,@^2_{\leq}\,0\,n\,h)\;.$$

□

Lemma 4. $\vdash \forall x\colon Nat\,.\,\forall y\colon Nat\,.\,Q(x,y)$.

Proof. By application of L_1. If $L_2\colon \forall x\colon Nat\,.\,\forall y\colon Nat\,.\,Q(x,y)$, then

$$L_2 =_{df} \lambda x\,.\,\lambda y\,.\,L_1\,(x+y)\;.$$

□

4.6 Modified ∀∃ Formula

In this final section, there is brief discussion of a modified form of the ∀∃ formula. Consider a model in which A is related to B via R, as shown in Figure 4.6.

$$A \overset{R}{\text{———}} B$$

Figure 4.6: A one-valued bidirectional relation R between A and B.

Let a_1 and a_2 be objects of A, and let b_1 and b_2 be objects of B. Now, it is a requirement of a bidirectional relation (previously unstated) that if a_1 is linked to b_1, then b_1 is linked to a_1, as in Figure 4.7 (left).

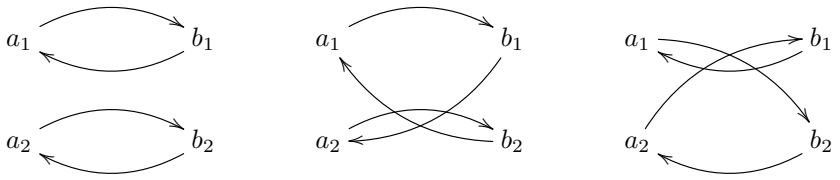

Figure 4.7: Three ways of linking objects of A and B in the absence of any constraints.

However, if the usual introduction rules of A and B apply, i.e.

$$\frac{b\colon B}{@_A\, b\colon A}\ (A\,I) \qquad\qquad \frac{a\colon A}{@_B\, a\colon B}\ (B\,I)\ ,$$

there is nothing to preclude a_1 from being linked to b_1, and b_1 from being linked to a_2, as shown in Figure 4.7 (centre), and reinforced by the `CoFixpoint` below.

Coq Listing 37. *(Irreferential Objects)* Objects `a1` and `b1` are irreferential, and so are `a2` and `b2`.

```
CoInductive A : Set :=
   Build_A : B -> A
with B : Set :=
   Build_B : A -> B.

CoFixpoint a1 : A :=
   Build_A b1
with a2 : A :=
   Build_A b2
with b1 : B :=
   Build_B a2
with b2 : B :=
   Build_B a1.
```

Clearly, a_1 is correctly linked to b_1 if and only if

$$R_B\, a_1 = b_1\ \wedge R_A\, b_1 = a_1\ ,$$

where R_A and R_B are relation functions on B and A respectively.

Definition 36. *(Bidirectional Links)* Let R be a bidirectional relation between classes A and B. If the multiplicity of the relation at B is one, R is correctly linked if

$$\forall a\colon A\,.\,R_A\,(R_B\,a) = a\ .$$

Alternatively, if the multiplicity of the relation at B is many, R is correctly linked if

$$\forall a\colon A\,.\,\forall b\colon B\,.\,b \in R_B\,a \to R_A\,b = a\ .$$

4.6.1 Preconditions

In the light of these considerations, it seems appropriate to modify the ∀∃ formula so that it rejects badly constructed source models, not least because the postcondition of a transformation may wish to navigate a bidirectional relation in both directions.

Definition 37. *(∀∃ Formula)* The specification of a transformation between A and B, which is subject to a precondition on A and a postcondition on A and B, is given by

$$\forall x\colon A\,.\,Pre\ a \to \exists y\colon B\,.\,Post\ x\ y\ .$$

Further, the certified program which inhabits it is

$$\lambda x\,.\,\lambda h\,.\,\langle (f\ x),\ q\rangle\ ,$$

where h is a proof that the precondition holds. Clearly, the program cannot be executed unless a suitable value of h is found.

In addition to checking the integrity of source models, preconditions play several other roles in model transformations. First, to allow a choice of rules in different cases, e.g. by checking that a class is a root class, if root classes are transformed by a different rule to non-root classes. Second, to ensure that a postcondition is well-defined, e.g. by insisting that $x \geq 0$ if the postcondition takes the square root of x. Third, to ensure that only certain source elements are transformed, e.g. by checking that a class is persistent, if only persistent classes are mapped to database tables. In the first and second cases, one might expect the precondition to contribute to the proof of the postcondition.

5
Ordered Model Transformations

This chapter defines a particular kind of model transformation in which the source and target models are ordered by containment. For pedagogical reasons, it starts with an informal definition in the language of sets. However, once a suitable example is in place, the remaining bulk of this chapter is taken up with a formal definition in the theory of types. A concrete example rounds off the chapter.

5.1 Introduction

By way of introduction, this chapter starts with an informal set theoretical definition of an ordered model transformation, which is based on the notion of an ordered class model $(C, <_C)$, as defined in Section 3.5. In due course, it will be replaced by a formal type theoretical definition.

Definition 38. *(Ordered Model Transformation - Set)* An ordered model transformation $(Tran, <_{Tran})$ is an irreflexive and transitive order relation $<_{Tran}$, over a set $Tran$ of transformations between the classes of an ordered source model $(Src, <_{Src})$ and an ordered target model $(Tgt, <_{Tgt})$, in which

- $Tran$ is a set of triples $(x \mapsto y, p, q)$ where
  - $x \in Src$ and $y \in Tgt$;
  - p is a predicate function on x that defines the precondition, and q is a predicate function on x and y that defines the postcondition, of a transformation $x \mapsto y$ between x and y, see Figure 5.1;

$$x \overset{p}{\vdash} - - - - \overset{q}{\succ} y$$

Figure 5.1: An element t_{xy} of $Tran$.

- $<_{Tran}$ is a set of pairs (t_{xy}, t_{zw}), where
  - $t_{xy}, t_{zw} \in Tran$;
  - $t_{xy} <_{Tran} t_{zw}$ means "t_{xy} is nested within t_{zw}", or alternatively "t_{zw} contains t_{xy}";

– $(x, z) \in <_{Src}$ and $(y, w) \in <_{Tgt}$;
– the multiplicities of the relations between z and x, and w and y, are the same, see Figure 5.2.

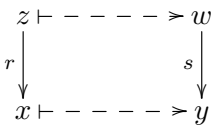

Figure 5.2: An element of $<_{Tran}$. The multiplicities of relations r and s are either both one or both many.

Definition 39. *(Totally Ordered Model Transformation - Set)* A totally ordered model transformation $(Tran, <_{Tran})$ is an ordered model transformation in which for all $t_1, t_2, t_3 \in Tran$,

$$\begin{aligned}
&\text{if } t_1 <_{Tran} t_2 \text{ and } t_2 <_{Tran} t_1 \text{ then } t_1 = t_2 \ ; \\
&\text{if } t_1 <_{Tran} t_2 \text{ and } t_2 <_{Tran} t_3 \text{ then } t_1 <_{Tran} t_3 \ ; \\
&\text{either } t_1 <_{Tran} t_2 \text{ or } t_2 <_{Tran} t_1 \ .
\end{aligned}$$

A totally ordered model transformation has the appearance of a ladder, in which each rung is a transformation between a source class and a target class, and each riser is a containment relation between a class and its parent. A partially ordered model transformation has the appearance of a tree of ladders.

Example 13. Figure 5.3 shows a partially ordered model transformation $(Tran, <_{Tran})$ between the classes of an ordered source model $(Src, <_{Src})$ and the classes of an ordered target model $(Tgt, <_{Tgt})$, where

$$\begin{aligned}
Src &=_{df} \{A, B, C, D\} \\
<_{Src} &=_{df} \{(B, A), (C, B), (D, B)\} \ ,
\end{aligned}$$

and

$$\begin{aligned}
Tgt &=_{df} \{W, X, Y, Z\} \\
<_{Tgt} &=_{df} \{(X, W), (Y, X), (Z, X)\} \ ,
\end{aligned}$$

and

$$\begin{aligned}
Tran &=_{df} \{t_{AW}, t_{BX}, t_{CY}, t_{DZ}\} \\
<_{Tran} &=_{df} \{(t_{BX}, t_{AW}), (t_{CY}, t_{BX}), (t_{DZ}, t_{BX})\} \ ,
\end{aligned}$$

where $t_{AW} = (A \mapsto W, Pre_A, Post_W)$, $t_{BX} = (B \mapsto X, Pre_B, Post_X)$ and so on, for some preconditions $Pre_A, \ldots, Pre_D$, and postconditions $Post_W, \ldots, Post_Z$. □

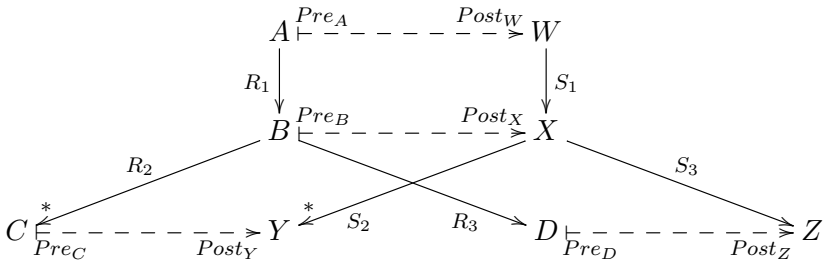

Figure 5.3: An ordered model transformation between an ordered source model $(Src, <_{Src})$ and an ordered target model $(Tgt, <_{Tgt})$.

5.2 Components

The fundamental components of an ordered model transformation $(T, <_T)$ are clearly the elements of T and $<_T$, and any type theoretical encoding of $(T, <_T)$ would inevitably be based upon them. A naive encoding of $(T, <_T)$ would see T and $<_T$ encoded as lists, as in Listing 38. However, while this kind of encoding is suitable for studying the properties of ordered model transformations in particular, it is not suitable for studying them in general, not least because it partially eschews the fact that $(T, <_T)$ is hierarchical.

Coq Listing 38. *(Literal Encoding)* The key fragments of a literal encoding of Example 13 are shown below. Compare and contrast the element definitions with those of $Tran$ and $<_{Tran}$ above. Note that `LA` (in the definition of `Tran` below) is a proof that $A \in Src$, `LAW` (in the definition of `Less_Tran` below) is a proof that $(A \mapsto W, Pre_A, Post_W) \in Tran$, and `LAB` (also in the definition of `Less_Tran` below) is a proof that $(B, A) \in <_{Src}$. Similarly for the other `L` terms.

```
Inductive Tran_Element : Type :=
   Build_Tran_Element :
      forall x : Set,
      In x Src ->
      forall y : Set,
      In y Tgt ->
      (x -> Prop) ->
      (x -> y -> Prop) ->
         Tran_Element.

Definition Tran : list Tran_Element :=
   Build_Tran_Element A LA W LW Pre_A Post_W ::
   Build_Tran_Element B LB X LX Pre_B Post_X ::
   Build_Tran_Element C LC Y LY Pre_C Post_Y ::
   Build_Tran_Element D LD Z LZ Pre_D Post_Z ::
   nil.

Inductive Less_Tran_Element : Type :=
   Build_Less_Tran_Element :
```

```
      forall x : Set,
      forall ix : In x Src,
      forall y : Set,
      forall jy : In y Tgt,
      forall px : x -> Prop,
      forall qy : x -> y -> Prop,
      In (Build_Tran_Element x ix y jy px qy) Tran ->

      forall z : Set,
      forall iz : In z Src,
      forall w : Set,
      forall jw : In w Tgt,
      forall pz : z -> Prop,
      forall qw : z -> w -> Prop,
      In (Build_Tran_Element z iz w jw pz qw) Tran ->

      In (Build_Less_Model_Element z x) Less_Src ->
      In (Build_Less_Model_Element w y) Less_Tgt ->
         Less_Tran_Element.

Definition Less_Tran : list Less_Tran_Element :=
   Build_Less_Tran_Element
      D LD Z LZ Pre_D Post_Z LDZ
      B LB X LX Pre_B Post_X LBX
      LBD LXZ ::
   Build_Less_Tran_Element
      C LC Y LY Pre_C Post_Y LCY
      B LB X LX Pre_B Post_X LBX
      LBC LXY ::
   Build_Less_Tran_Element
      B LB X LX Pre_B Post_X LBX
      A LA W LW Pre_A Post_W LAW
      LAB LWX ::
   nil.
```

This completes the introductory set theoretical description of an ordered model transformation. In the following sections, a suitably robust type theoretical encoding of $(T, <_T)$ is given by five dependent types: *Data*, *Link*, *DataLink*, *Poset* and *Tran*. Briefly, *Data* is an encoding of an element of T, *Link* is an encoding of an element of $<_T$; and through *DataLink* and *Poset*, *Tran* is an encoding of $(T, <_T)$.

5.2.1 Data Component

A data component is an ordered 4-tuple $(X, Y, Pre, Post)$, in which Pre defines the precondition and $Post$ defines the postcondition of a transformation between the base attributes of a source class X (and if appropriate, those of any classes that directly or indirectly contain X), and the base attributes of a target class Y, see Figure 5.4. Since a class is generally characterised by both base and referential attributes, a transformation between X and Y

$$X \overset{Pre}{\vdash\!\!-\!\!-\!\!-\!\!-\!\!-\!\!-\!\!\rightarrow} Y$$

Figure 5.4: A data component on X and Y. X may be a root class, an intermediate class, or a leaf class. Similarly for Y.

actually comprises two transformations: a transformation between the base attributes of X and Y (as captured by a data component), and an orthogonal transformation between the referential attributes of X and Y (as captured by an allied link component, and described in the next section). In terms of the ladder analogy, a data component captures the essential characteristics of a transformation between the ends of a rung. Note that every ordered model transformation has at least one data component, i.e. the data component on the root classes.

Definition 40. *(Data Component)* A data component on a source class X and a target class Y is an inhabitant of type $Data\, X\, Y$.

Formation Rule

The *Data* formation rule, which is given by

$$\frac{X : U_0 \quad Y : U_0}{Data\, X\, Y : U_1}\ (Data\, F)\ ,$$

asserts that $Data\, X\, Y$ is an inhabitant of U_1, if X and Y are inhabitants of U_0. Since by definition all classes inhabit U_0, the conclusion always holds.

Introduction Rule

The *Data* introduction rule, which is given by

$$\frac{\begin{array}{ll} X : U_0 & Y : U_0 \end{array} \qquad \begin{array}{l} Pre : X \to U_0 \\ Post : X \to Y \to U_0 \end{array}}{@_{Data}\, X\, Y\, Pre\, Post : Data\, X\, Y}\ (Data\, I)\ ,$$

asserts that $(@_{Data}\, X\, Y\, Pre\, Post)$ is a data component on X and Y, if Pre is a predicate function on X, and $Post$ is a predicate function on X and Y. In Figure 5.3, the data component d_{AW} on A and W is given by

$$\frac{\begin{array}{ll} A : U_0 & W : U_0 \end{array} \qquad \begin{array}{l} Pre_A : A \to U_0 \\ Post_W : A \to W \to U_0 \end{array}}{(d_{AW} =_{df} @_{Data}\, A\, W\, Pre_A\, Post_W) : Data\, A\, W}\ (Data\, I)\ .$$

Similarly,

$$\begin{array}{l} (d_{BX} =_{df} @_{Data}\, B\, X\, Pre_B\, Post_X) : Data\, B\, X \\ (d_{CY} =_{df} @_{Data}\, C\, Y\, Pre_C\, Post_Y) : Data\, C\, Y \\ (d_{DZ} =_{df} @_{Data}\, D\, Z\, Pre_D\, Post_Z) : Data\, D\, Z\ . \end{array}$$

Elimination Rules

The *Data* elimination rules are given by

$$\frac{\begin{array}{l}P\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,(Data\;X\;Y \to U_n)\\ X\colon U_0\\ Y\colon U_0\\ d\colon Data\;X\;Y\\ i\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall Pre\colon X \to U_0\,.\,\forall Post\colon X \to Y \to U_0\,.\\ \qquad P\;X\;Y\;(@_{Data}\;X\;Y\;Pre\;Post)\end{array}}{@^{-1}_{Data}\;P\;X\;Y\;d\;i\colon P\;X\;Y\;d}\;(Data\,E_n)\;,$$

for $n = 0, 1, 2, \ldots$. Each rule asserts, for a particular universe U_n, that $P\,X\,Y\,d$ is inhabited for all data components d, if $P\,X\,Y\;(@_{Data}\;X\;Y\;Pre\;Post)$ is inhabited for all preconditions Pre and postconditions $Post$ (this is a rather subtle way of stating that $@_{Data}$ is the *only* constructor of $Data\;X\;Y$). Discharging the premisses gives

$$\phi_{Data} \;=_{df}\; \lambda P\,.\,\lambda i\,.\,\lambda X\,.\,\lambda Y\,.\,\lambda d\,.\,@^{-1}_{Data}\;P\;X\;Y\;d\;i$$

is an inhabitant of type

$$\begin{array}{c}\forall P\colon (\forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,(Data\;X\;Y \to U_n))\,.\\ (\forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall Pre\colon X \to U_0\,.\,\forall Post\colon X \to Y \to U_0\,.\\ P\;X\;Y\;(@_{Data}\;X\;Y\;Pre\;Post)) \to\\ \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall d\colon Data\;X\;Y\,.\,P\;X\;Y\;d\;.\end{array}$$

Computation Rule

The *Data* computation rule, which is given by

$$@^{-1}_{Data}\;P\;X\;Y\;(@_{Data}\;X\;Y\;Pre\;Post)\;i \twoheadrightarrow i\;X\;Y\;Pre\;Post\;,$$

asserts that the term on the right hand side is a simplification of the term on the left hand side, in the sense that it is structurally smaller. However, both terms have the same type, namely $P\;X\;Y\;(@_{Data}\;X\;Y\;Pre\;Post)$.

Coq Listing 39. *(Data)* The encoding of a data component in Coq not only defines the formation and introduction rules, but it also contains sufficient information to enable Coq to automatically define the elimination and computation rules.

```
Inductive Data : Set -> Set -> Type :=
   Build_Data :
      forall X : Set,
      forall Y : Set,
      (X -> Prop) ->
      (X -> Y -> Prop) ->
         Data X Y.
```

Compare and contrast Coq's rendition of the *Data* elimination and computation rules below, with ϕ_{Data} and the *Data* computation rule above.

```
Data_rect =
fun (P : forall P P0 : Set, Data P P0 -> Type)
  (f : forall (X Y : Set) (P0 : X -> Prop) (P1 : X -> Y -> Prop),
       P X Y (Build_Data X Y P0 P1)) (P0 P1 : Set) (d : Data P0 P1) =>
match d as d0 in (Data P2 P3) return (P P2 P3 d0) with
| Build_Data x x0 x1 x2 => f x x0 x1 x2
end
     : forall P : forall P P0 : Set, Data P P0 -> Type,
       (forall (X Y : Set) (P0 : X -> Prop) (P1 : X -> Y -> Prop),
        P X Y (Build_Data X Y P0 P1)) ->
       forall (P0 P1 : Set) (d : Data P0 P1), P P0 P1 d
```

5.2.2 Link Component

Consider an ordered model transformation in which a source class X is transformed into a target class Y, and a source class X' (a child of X) is transformed into a target class Y' (a child of Y). A link component of t (see Figure 5.5) is an ordered 6-tuple (X, Y, X', Y', R, S), in which R is a relation between X and X' (a referential attribute of X, in other words), and S is a relation between Y and Y' (a referential attribute of Y). In terms of the ladder analogy, a link component captures the essential characteristics of a transformation between a source riser and a target riser; it plays no part in capturing the characteristics of the transformations between the ends of the rungs that define the risers, because these are the province of data components.

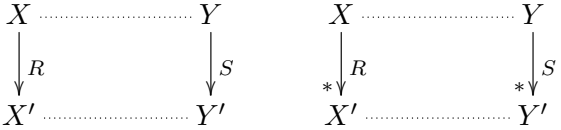

Figure 5.5: Link components on X, Y, X' and Y'. The mutliplicities of R and S in each case are the same, i.e. both one-valued or both many-valued.

Definition 41. *(Link Component)* A link component on source classes X and X', and target classes Y and Y', is an inhabitant of type $Link\ X\ Y\ X'\ Y'$.

Formation Rule

The *Link* formation rule, which is given by

$$\frac{X : U_0 \quad Y : U_0 \quad X' : U_0 \quad Y' : U_0}{Link\ X\ Y\ X'\ Y' : U_1}\ (Link\ F)\ ,$$

asserts that $Link\ X\ Y\ X'\ Y'$ is an inhabitant of U_1, if X, Y, X' and Y' are inhabitants of U_0.

Introduction Rules

There are two *Link* introduction rules. The first rule, which is given by

$$\frac{\begin{array}{cccc} X\colon U_0 & Y\colon U_0 & X'\colon U_0 & Y'\colon U_0 \end{array} \quad \begin{array}{l} R\colon X \to X' \\ S\colon Y \to Y' \end{array}}{@^1_{Link}\, X\ Y\ X'\ Y'\ R\ S\colon Link\ X\ Y\ X'\ Y'}\ (Link\, I_1)\ ,$$

asserts that $(@^1_{Link}\ X\ Y\ X'\ Y'\ R\ S)$ is a link component on X, Y, X' and Y', if R is a relation that takes X to X', and S is a relation that takes Y to Y', i.e. if the multiplicities of R and S are both one. The second rule, which is given by

$$\frac{\begin{array}{cccc} X\colon U_0 & Y\colon U_0 & X'\colon U_0 & Y'\colon U_0 \end{array} \quad \begin{array}{l} R\colon X \to [\,X'\,] \\ S\colon Y \to [\,Y'\,] \end{array}}{@^2_{Link}\, X\ Y\ X'\ Y'\ R\ S\colon Link\ X\ Y\ X'\ Y'}\ (Link\, I_2)\ ,$$

asserts that $(@^2_{Link} X\ Y\ X'\ Y'\ R\ S)$ is a link component on X, Y, X' and Y', if R is a relation that takes X to $[\,X'\,]$, and S is a relation that takes Y to $[\,Y'\,]$, i.e. if the multiplicities of R and S are both many. In Figure 5.3, the link component l_{AW} on A, W, B and X is given by

$$\frac{\begin{array}{cccc} A\colon U_0 & W\colon U_0 & B\colon U_0 & X\colon U_0 \end{array} \quad \begin{array}{l} R_1\colon A \to B \\ S_1\colon W \to X \end{array}}{(l_{AW} =_{df} @^1_{Link}\, A\ W\ B\ X\ R_1\ S_1)\colon Link\ A\ W\ B\ X}\ (Link\, I_1)\ .$$

Similarly,

$$\begin{aligned} (l^1_{BX} &=_{df} @^2_{Link}\, B\ X\ C\ Y\ R_2\ S_2)\colon Link\ B\ X\ C\ Y \\ (l^2_{BX} &=_{df} @^1_{Link}\, B\ X\ D\ Z\ R_3\ S_3)\colon Link\ B\ X\ D\ Z\ . \end{aligned}$$

Elimination Rules

The *Link* elimination rules are given by

$$\frac{\begin{array}{l} P\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,(Link\ X\ Y\ X'\ Y' \to U_n) \\ X\colon U_0 \\ Y\colon U_0 \\ X'\colon U_0 \\ Y'\colon U_0 \\ l\colon Link\ X\ Y\ X'\ Y' \\ i\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,\forall R\colon X \to X'\,.\,\forall S\colon Y \to Y'\,. \\ \qquad P\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ R\ S) \\ j\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,\forall R\colon X \to [\,X'\,]\,.\,\forall S\colon Y \to [\,Y'\,]\,. \\ \qquad P\ X\ Y\ X'\ Y'\ (@^2_{Link}\ X\ Y\ X'\ Y'\ R\ S) \end{array}}{@^{-1}_{Link}\, P\ X\ Y\ X'\ Y'\ l\ i\ j\colon P\ X\ Y\ X'\ Y'\ l}\ (Link\, E_n)\ ,$$

for $n = 0, 1, 2, \ldots$. Each rule asserts, for a particular universe U_n, that $P\ X\ Y\ X'\ Y'\ l$ is inhabited for all link components l, if $P\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ R\ S)$ is inhabited for

all relations R and S with a multiplicity of one, and $P\,X\,Y\,X'\,Y'\,(@^2_{Link}\,X\,Y\,X'\,Y'\,R\,S)$ is inhabited for all relations R and S with a multiplicity of many.

Computation Rules

The *Link* computation rules are given by

$$@^{-1}_{Link}\,P\,X\,Y\,X'\,Y'\,(@^1_{Link}\,X\,Y\,X'\,Y'\,R\,S)\,i\,j \twoheadrightarrow i\,X\,Y\,X'\,Y'\,R\,S$$
$$@^{-1}_{Link}\,P\,X\,Y\,X'\,Y'\,(@^2_{Link}\,X\,Y\,X'\,Y'\,R\,S)\,i\,j \twoheadrightarrow j\,X\,Y\,X'\,Y'\,R\,S\,.$$

Each rule asserts that the term on the right hand side is a simplification of the term on the left hand side, in the sense that it is structurally smaller.

Coq Listing 40. *(Link)* The encoding of a link component in Coq is shown below. Note that there is a separate constructor for each kind of relation.

```
Inductive Link : Set -> Set -> Set -> Set -> Type :=
   Build_Link_1 :
      forall X : Set,
      forall Y : Set,
      forall X' : Set,
      forall Y' : Set,
      (X -> X') ->
      (Y -> Y') ->
         Link X Y X' Y'|
   Build_Link_2 :
      forall X : Set,
      forall Y : Set,
      forall X' : Set,
      forall Y' : Set,
      (X -> list X') ->
      (Y -> list Y') ->
         Link X Y X' Y'.
```

5.2.3 Data Link Component

A data link component is the composition of a data component and a link component, in a U-shaped configuration that has sides and a bottom but no top, see Figure 5.6. It is the fundamental composite structure from which ordered model transformations are constructed. In terms of the ladder analogy, a data link component captures the essential characteristics of two transformations: a transformation between the ends of a rung, and a transformation between the risers at the ends of the rung.

Definition 42. *(DataLink)* A data link component on source classes X and X', and target classes Y and Y', is an inhabitant of type $DataLink\,X\,Y\,X'\,Y'$.

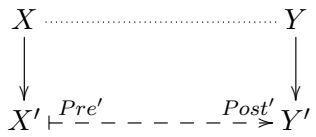

Figure 5.6: A U-shaped data link component on X, Y, X' and Y', with sides and a bottom but no top.

Formation Rule

The *DataLink* formation rule, which is given by

$$\frac{X\colon U_0 \quad Y\colon U_0 \quad X'\colon U_0 \quad Y'\colon U_0}{DataLink\ X\ Y\ X'\ Y'\colon U_1}\ (DataLink\ F)\ ,$$

asserts that $DataLink\,X\,Y\,X'\,Y'$ is an inhabitant of U_1, if X, Y, X' and Y' are inhabitants of U_0.

Introduction Rule

The *DataLink* introduction rule, which is given by

$$\frac{\begin{array}{cccc} X\colon U_0 & Y\colon U_0 & X'\colon U_0 & Y'\colon U_0 \end{array} \quad \begin{array}{l} d'\colon Data\ X'\ Y' \\ l\colon Link\ X\ Y\ X'\ Y' \end{array}}{@_{DataLink}\ X\ Y\ X'\ Y'\ d'\ l\colon DataLink\ X\ Y\ X'\ Y'}\ (DataLink\ I)\ ,$$

asserts that $(@_{DataLink}\ X\ Y\ X'\ Y'\ d'\ l)$ is a data link component on X, Y, X' and Y', if d' is a data component on X' and Y', and l is a link component on X, Y, X' and Y'. In Figure 5.3, the data link component dl_{AW} on A, W, B and X is given by

$$\frac{\begin{array}{cccc} A\colon U_0 & W\colon U_0 & B\colon U_0 & X\colon U_0 \end{array} \quad \begin{array}{l} d_{BX}\colon Data\ B\ X \\ l_{AW}\colon Link\ A\ W\ B\ X \end{array}}{(dl_{AW}\ =_{df}\ @_{DataLink}\ A\ W\ B\ X\ d_{BX}\ l_{AW})\colon DataLink\ A\ W\ B\ X}\ (DataLink\ I)\ ,$$

where d_{BX} and l_{AW} are as defined in Sections 5.2.1 and 5.2.2 respectively. Similarly,

$$(dl^1_{BX}\ =_{df}\ @_{DataLink}\ B\ X\ C\ Y\ d_{CY}\ l^1_{BX})\colon DataLink\ B\ X\ C\ Y$$
$$(dl^2_{BX}\ =_{df}\ @_{DataLink}\ B\ X\ D\ Z\ d_{DZ}\ l^2_{BX})\colon DataLink\ B\ X\ D\ Z\ .$$

Elimination Rules

The *DataLink* elimination rules are given by

$$\frac{\begin{array}{l} P\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,(\mathit{DataLink}\,X\,Y\,X'\,Y' \to U_n) \\ X\colon U_0 \\ Y\colon U_0 \\ X'\colon U_0 \\ Y'\colon U_0 \\ dl\colon \mathit{DataLink}\,X\,Y\,X'\,Y' \\ i\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,. \\ \qquad \forall d'\colon \mathit{Data}\,X'\,Y'\,.\,\forall l\colon \mathit{Link}\,X\,Y\,X'\,Y'\,. \\ \qquad\qquad P\,X\,Y\,X'\,Y'\,(@_{\mathit{DataLink}}\,X\,Y\,X'\,Y'\,d'\,l) \end{array}}{@^{-1}_{\mathit{DataLink}}\,P\,X\,Y\,X'\,Y'\,dl\,i\colon P\,X\,Y\,X'\,Y'\,dl}\ (\mathit{DataLink}\,E_n)\ ,$$

for $n = 0, 1, 2, \ldots$. Each rule asserts, for a particular universe U_n, that $P\,X\,Y\,X'\,Y'\,dl$ is inhabited for all data link components dl, if $P\,X\,Y\,X'\,Y'\,(@_{\mathit{DataLink}}\,X\,Y\,X'\,Y'\,d'\,l)$ is inhabited for all data components d', and all link components l.

Computation Rule

The *DataLink* computation rule, which is given by

$$@^{-1}_{\mathit{DataLink}}\,P\,X\,Y\,X'\,Y'\,(@_{\mathit{DataLink}}\,X\,Y\,X'\,Y'\,d'\,l)\,i \twoheadrightarrow i\,X\,Y\,X'\,Y'\,d'\,l\ ,$$

asserts that the term on the right hand side is a simplification of the term on the left hand side, in the sense that it is structurally smaller. However, both terms have the same type, i.e.

$$P\,X\,Y\,X'\,Y'\,(@_{\mathit{DataLink}}\,X\,Y\,X'\,Y'\,d'\,l)\ .$$

Coq Listing 41. *(DataLink)* The encoding of a data link component is shown below. Note that the constructor requires a data component and a link component to construct a data link component.

```
Inductive DataLink : Set -> Set -> Set -> Set -> Type :=
   Build_DataLink :
      forall X : Set,
      forall Y : Set,
      forall X' : Set,
      forall Y' : Set,
      Data X' Y' ->
      Link X Y X' Y' ->
         DataLink X Y X' Y'.
```

5.2.4 Data Link Poset

A data link poset is a non-empty and strict partially ordered set of data link components, in which order is defined by containment, as described in Section 5.2.5. In terms of the

ladder analogy, a data link poset is one rung short of a fully-formed tree of ladders, which captures the essential characteristics of every transformation between the risers and every transformation between the ends of the rungs, apart from the one between the ends of the root rung, the rung at the top of the ladder. The root transformation, which is described in Section 5.2.6, is handled separately.

Figure 5.7 shows three data link posets on X and Y, featuring from left to right: a solitary data link component on X, Y, X' and Y'; two data link components on X, Y, X' and Y', and X', Y', X'' and Y'', one atop (and therefore greater than) the other; two data link components on X, Y, X' and Y', and X, Y, X'' and Y'', each beside (and therefore neither greater than nor less than) the other.

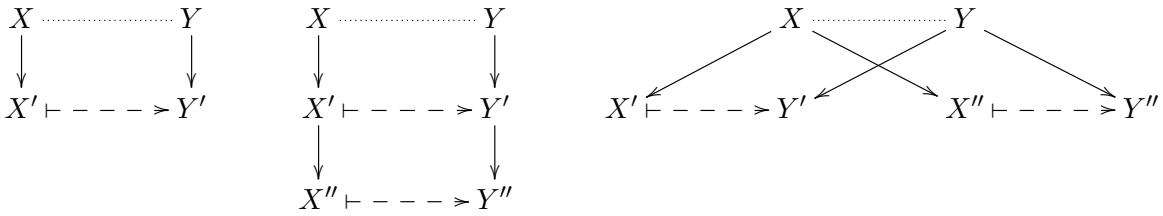

Figure 5.7: Three data link posets on X and Y.

Definition 43. *(Poset)* A data link poset on a source class X and a target class Y is an inhabitant of type $Poset\, X\, Y$.

Format Rule

The *Poset* formation rule, which is given by

$$\frac{X : U_0 \quad Y : U_0}{Poset\, X\, Y : U_1}\ (Poset\, F)\ ,$$

asserts that $Poset\, X\, Y$ is an inhabitant of U_1, if X and Y are inhabitants of U_0.

Introduction Rules

There are three *Poset* introduction rules: *base*, *step* and *join*. Each one is depicted in Figure 5.7. The *base* rule (left figure), which is given by

$$\frac{X : U_0 \quad Y : U_0 \quad X' : U_0 \quad Y' : U_0 \quad dl : DataLink\, X\, Y\, X'\, Y'}{@^1_{Poset}\, X\, Y\, X'\, Y'\, dl : Poset\, X\, Y}\ (Poset\, I_1)\ ,$$

asserts that $(@^1_{Poset}\, X\, Y\, X'\, Y'\, dl)$ is a data link poset on X and Y, if dl is a data link component on X, Y, X', and Y'. In Figure 5.3, the data link poset p^1_{BX} on B and X is given by

$$\frac{B : U_0 \quad X : U_0 \quad C : U_0 \quad Y : U_0 \quad dl^1_{BX} : DataLink\, B\, X\, C\, Y}{(p^1_{BX} =_{df} @^1_{Poset}\, B\, X\, C\, Y\, dl^1_{BX}) : Poset\, B\, X}\ (Poset\, I_1)\ ,$$

where dl^1_{BX} is as defined in Section 5.2.3. Similarly,

$$(p^2_{BX} =_{df} @^1_{Poset}\, B\, X\, D\, Z\, dl^2_{BX}) \colon Poset\, B\, X\ .$$

Remark 9. One could regard p^1_{BX} as either a root transformation short of an ordered model transformation on B and X, or simply part of an ordered model transformation on A and W. The same could be said of the other inhabitants of $Poset\, B\, X$. □

The *step* rule (central figure), which is given by

$$\frac{\begin{array}{cccc} X \colon U_0 & Y \colon U_0 & X' \colon U_0 & Y' \colon U_0 \end{array} \quad \begin{array}{l} dl \colon DataLink\, X\, Y\, X'\, Y' \\ p' \colon Poset\, X'\, Y' \end{array}}{@^2_{Poset}\, X\, Y\, X'\, Y'\, dl\, p' \colon Poset\, X\, Y}\ (Poset\, I_2)\ ,$$

asserts that $(@^2_{Poset}\, X\, Y\, X'\, Y'\, dl\, p')$ is a data link poset on X and Y, if dl is a data link component on X, Y, X', and Y', and p' is a data link poset on X' and Y'. In Figure 5.3, the data link poset p_{AW} on A and W is given by

$$\frac{\begin{array}{cccc} A \colon U_0 & W \colon U_0 & B \colon U_0 & X \colon U_0 \end{array} \quad \begin{array}{l} dl_{AW} \colon DataLink\, A\, W\, B\, X \\ p^3_{BX} \colon Poset\, B\, X \end{array}}{(p_{AW} =_{df} @^2_{Poset}\, A\, W\, B\, X\, dl_{AW}\, p^3_{BX}) \colon Poset\, A\, W}\ (Poset\, I_2)\ ,$$

where dl_{AW} is as defined in Section 5.2.3, and p^3_{BX} is defined below. Finally, the *join* rule (right figure), which is given by

$$\frac{\begin{array}{cc} X \colon U_0 & Y \colon U_0 \end{array} \quad \begin{array}{l} p_1 \colon Poset\, X\, Y \\ p_2 \colon Poset\, X\, Y \end{array}}{@^3_{Poset}\, X\, Y\, p_1\, p_2 \colon Poset\, X\, Y}\ (Poset\, I_3)\ ,$$

asserts that $(@^3_{Poset}\, X\, Y\, X'\, Y'\, p_1\, p_2)$ is a data link poset on X and Y, if p_1 and p_2 are data link posets on X and Y. In Figure 5.3, the data link poset p^3_{BX} on B and X is given by

$$\frac{\begin{array}{cc} B \colon U_0 & X \colon U_0 \end{array} \quad \begin{array}{l} p^1_{BX} \colon Poset\, B\, X \\ p^2_{BX} \colon Poset\, B\, X \end{array}}{(p^3_{BX} =_{df} @^3_{Poset}\, B\, X\, p^1_{BX}\, p^2_{BX}) \colon Poset\, B\, X}\ (Poset\, I_3)\ .$$

Elimination Rules

The *Poset* elimination rules are given by

$$\frac{\begin{array}{l} P\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,(Poset\;X\;Y \to U_n) \\ X\colon U_0 \\ Y\colon U_0 \\ p\colon Poset\;X\;Y \\ i\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,\forall dl\colon DataLink\;X\;Y\;X'\;Y'\,. \\ \qquad P\;X\;Y\;(@^1_{Poset}\;X\;Y\;X'\;Y'\;dl) \\ j\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,\forall dl\colon DataLink\;X\;Y\;X'\;Y'\,. \\ \qquad \forall p'\colon Poset\;X'\;Y'\,.\,P\;X'\;Y'\;p' \to \\ \qquad\qquad P\;X\;Y\;(@^2_{Poset}\;X\;Y\;X'\;Y'\;dl\;p') \\ k\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall p_1\colon Poset\;X\;Y\,.\,\forall p_2\colon Poset\;X\;Y\,. \\ \qquad P\;X\;Y\;p_1 \to P\;X\;Y\;p_2 \to P\;X\;Y\;(@^3_{Poset}\;X\;Y\;p_1\;p_2) \end{array}}{@^{-1}_{Poset}\;P\;X\;Y\;p\;i\;j\;k\colon P\;X\;Y\;p}\;(Poset\,E_n)\;,$$

for $n = 0, 1, 2, \ldots$. Each rule asserts, for a particular universe U_n, that $P\,X\,Y\,p$ is inhabited for all data link posets p, if

- $P\;X\;Y\;(@^1_{Poset}\;X\;Y\;X'\;Y'\;dl)$ is inhabited for all data link component dl;
- $P\;X\;Y\;(@^2_{Poset}\;X\;Y\;X'\;Y'\;dl\;p')$ is inhabited for all data link components dl, and all data link posets p', if $P\;X\;Y p'$ is inhabited for all data link posets p'; and
- $P\;X\;Y\;(@^3_{Poset}\;X\;Y\;p_1\;p_2)$ is inhabited for all data link posets p_1 and p_2, if $P\;X\;Y p_1$ and $P\;X\;Y p_2$ are inhabited for all data link posets p_1 and p_2.

It stands to reason that if $PX\;Y\;p$ is shown to be inhabited every which way that p can be constructed, then $PX\;Y\;p$ must be inhabited for all p.

Computation Rules

The *Poset* computation rules are given by

$$\begin{array}{l} @^{-1}_{Poset}\;P\;X\;Y\;(@^1_{Poset}\;X\;Y\;X'\;Y'\;dl)\;i\;j\;k \twoheadrightarrow i\;X\;Y\;X'\;Y'\;dl \\ @^{-1}_{Poset}\;P\;X\;Y\;(@^2_{Poset}\;X\;Y\;X'\;Y'\;dl\;p')\;i\;j\;k \twoheadrightarrow \\ \qquad j\;X\;Y\;X'\;Y'\;dl\;p'\;(@^{-1}_{Poset}\;P\;X'\;Y'\;p'\;i\;j\;k) \\ @^{-1}_{Poset}\;P\;X\;Y\;(@^3_{Poset}\;X\;Y\;p_1\;p_2)\;i\;j\;k \twoheadrightarrow \\ \qquad k\;X\;Y\;p_1\;p_2\;(@^{-1}_{Poset}\;P\;X\;Y\;p_1\;i\;j\;k)\;(@^{-1}_{Poset}\;P\;X\;Y\;p_2\;i\;j\;k)\;. \end{array}$$

Each rule asserts that the term on the right hand side is a simplification of the term on the left, in the sense that it is structurally smaller. Note that the second and third rules are recursive.

Coq Listing 42. *(Poset)* The encoding of a data link poset is shown below. Note that there are three constructors: the first requires a data link component; the second requires a data link component and a data link poset; and the third requires two data link posets.

```
Inductive Poset : Set -> Set -> Type :=
   Build_Poset_1 :
      forall X : Set,
      forall Y : Set,
      forall X' : Set,
      forall Y' : Set,
      DataLink X Y X' Y' ->
         Poset X Y |
   Build_Poset_2 :
      forall X : Set,
      forall Y : Set,
      forall X' : Set,
      forall Y' : Set,
      DataLink X Y X' Y' ->
      Poset X' Y' ->
         Poset X Y |
   Build_Poset_3 :
      forall X : Set,
      forall Y : Set,
      Poset X Y ->
      Poset X Y ->
         Poset X Y.
```

5.2.5 Order Relation $<_{DataLink}$

Informally, a data link poset is a partially ordered set of data link components, in which order is defined by containment in the sense that if dl_1 is a data link component on X_1, Y_1, X_1' and Y_1', and dl_2 is a data link component on X_2, Y_2, X_2' and Y_2', then the proposition "dl_1 is less than dl_2" holds if and only if X_2 is an ancestor of X_1 (a parent, in fact, if X_1 and X_2' are one and the same class), and Y_2 is an ancestor of Y_1. In terms of the ladder analogy, "dl_1 is less than dl_2" if a) the data component rung of dl_1 is below that of dl_2, and b) there is a path between the rungs of dl_1 and dl_2 which does not involve hopping between ladders.

Definition 44. *($<_{DataLink}$)* Let dl_1 be a data link component on X_1, Y_1, X_1' and Y_1', and let dl_2 be a data link component on X_2, Y_2, X_2' and Y_2'. If p is a proof that "dl_1 is less than dl_2", then p is an inhabitant of type

$$<_{DataLink} X_1\, Y_1\, X_1'\, Y_1'\, dl_1\, X_2\, Y_2\, X_2'\, Y_2'\, dl_2\ .$$

Formation Rule

The $<_{DataLink}$ formation rule, which is given by

$$\frac{\begin{array}{llll} X_1 : U_0 & Y_1 : U_0 & X_1' : U_0 & Y_1' : U_0 \quad dl_1 : DataLink\ X_1\ Y_1\ X_1'\ Y_1' \\ X_2 : U_0 & Y_2 : U_0 & X_2' : U_0 & Y_2' : U_0 \quad dl_2 : DataLink\ X_2\ Y_2\ X_2'\ Y_2' \end{array}}{<_{DataLink}\ X_1\ Y_1\ X_1'\ Y_1'\ dl_1\ X_2\ Y_2\ X_2'\ Y_2'\ dl_2 : U_1}\ (<_{DataLink} F)\ ,$$

asserts that $<_{DataLink}\ X_1\ Y_1\ X_1'\ Y_1'\ dl_1\ X_2\ Y_2\ X_2'\ Y_2'\ dl_2$ is an inhabitant of U_1, if X_1 through Y_2' are inhabitants of U_0, dl_1 is an inhabitant of $DataLink\ X_1\ Y_1\ X_1'\ Y_1'$, and dl_2 is an inhabitant of $DataLink\ X_2\ Y_2\ X_2'\ Y_2'$.

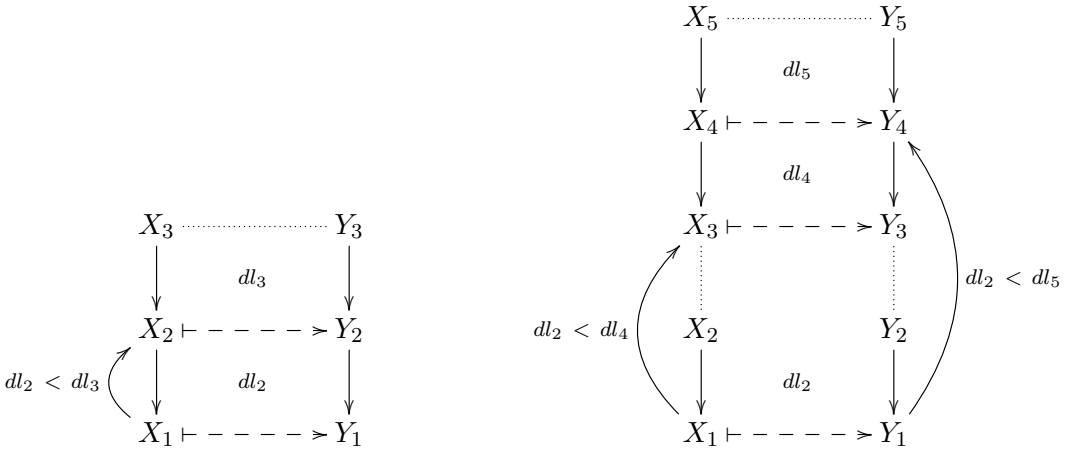

Figure 5.8: Illustrations of the $<_{DataLink}$ introduction rules.

Introduction Rules

There are two $<_{DataLink}$ introduction rules. The first rule, which is illustrated in Figure 5.8 (left) and given by

$$\frac{\begin{array}{llll} X_2 : U_0 & Y_2 : U_0 & X_1 : U_0 & Y_1 : U_0 \quad dl_2 : DataLink\ X_2\ Y_2\ X_1\ Y_1 \\ X_3 : U_0 & Y_3 : U_0 & dl_3 : DataLink\ X_3\ Y_3\ X_2\ Y_2 & \end{array}}{\begin{array}{l} @^1_{<_{DataLink}}\ X_2\ Y_2\ X_1\ Y_1\ dl_2\ X_3\ Y_3\ dl_3 : \\ \quad <_{DataLink}\ X_2\ Y_2\ X_1\ Y_1\ dl_2\ X_3\ Y_3\ X_2\ Y_2\ dl_3 \end{array}}\ (<_{DataLink} I_1)\ ,$$

asserts that

$$@^1_{<_{DataLink}}\ X_2\ Y_2\ X_1\ Y_1\ dl_2\ X_3\ Y_3\ dl_3$$

is a proof that "dl_2 is less than dl_3", if dl_3 is a successor of dl_2. For example,

$$@^1_{<_{DataLink}}\ B\ X\ C\ Y\ dl^1_{BX}\ A\ W\ dl_{AW}$$

is a proof that "dl^1_{BX} is less than dl_{AW}", where dl^1_{BX} and dl_{AW} are as defined in Section 5.2.3. The second rule, which is illustrated in Figure 5.8 (right) and given by

$$\frac{\begin{array}{l} X_2 : U_0 \quad Y_2 : U_0 \quad X_1 : U_0 \quad Y_1 : U_0 \quad dl_2 : DataLink\ X_2\ Y_2\ X_1\ Y_1 \\ X_4 : U_0 \quad Y_4 : U_0 \quad X_3 : U_0 \quad Y_3 : U_0 \quad dl_4 : DataLink\ X_4\ Y_4\ X_3\ Y_3 \\ p : <_{DataLink}\ X_2\ Y_2\ X_1\ Y_1\ dl_2\ X_4\ Y_4\ X_3\ Y_3\ dl_4 \\ X_5 : U_0 \quad Y_5 : U_0 \quad X_4 : U_0 \quad Y_4 : U_0 \quad dl_5 : DataLink\ X_5\ Y_5\ X_4\ Y_4 \end{array}}{\begin{array}{l} @^2_{<_{DataLink}}\ X_2\ Y_2\ X_1\ Y_1\ dl_2\ X_4\ Y_4\ X_3\ Y_3\ dl_4\ p\ X_5\ Y_5\ X_4\ Y_4\ dl_5 : \\ \quad <_{DataLink}\ X_2\ Y_2\ X_1\ Y_1\ dl_2\ X_5\ Y_5\ X_4\ Y_4\ dl_5 \end{array}}\ (<_{DataLink} I_2)\ ,$$

asserts that

$$@^2_{<_{DataLink}}\ X_2\ Y_2\ X_1\ Y_1\ dl_2\ X_4\ Y_4\ X_3\ Y_3\ dl_4\ p\ X_5\ Y_5\ X_4\ Y_4\ dl_5$$

is a proof that "dl_2 is less than dl_5", if p is a proof that "dl_2 is less than dl_4", and dl_5 is a successor of dl_4.

Elimination and Computation Rules

The $<_{DataLink}$ elimination and computation rules are intentionally undefined.

Coq Listing 43. *($<_{DataLink}$)* The encoding of $<_{DataLink}$ is shown below. Note that there are two constructors: the first constructor requires two data link components (the first of which is provided by the parameter `dl2`), and the second constructor requires three data link components. The second constructor also requires a proof that the first data link component `dl2` is less than the second data link component `dl4`.

```
Inductive LessDataLink
   (X2 : Set) (Y2 : Set) (X1 : Set) (Y1 : Set)
      (dl2 : DataLink X2 Y2 X1 Y1) :
   forall X : Set, forall Y : Set, forall X' : Set, forall Y' : Set,
      DataLink X Y X' Y' -> Prop :=

   Build_LessDataLink_1 :
      forall X3 : Set,
      forall Y3 : Set,
      forall dl3 : DataLink X3 Y3 X2 Y2,
         LessDataLink X2 Y2 X1 Y1 dl2 X3 Y3 X2 Y2 dl3 |

   Build_LessDataLink_2 :
      forall X4 : Set,
      forall Y4 : Set,
      forall X3 : Set,
      forall Y3 : Set,
      forall dl4 : DataLink X4 Y4 X3 Y3,
      LessDataLink X2 Y2 X1 Y1 dl2 X4 Y4 X3 Y3 dl4 ->
      forall X5 : Set,
      forall Y5 : Set,
      forall dl5 : DataLink X5 Y5 X4 Y4,
         LessDataLink X2 Y2 X1 Y1 dl2 X5 Y5 X4 Y4 dl5.
```

5.2.6 Ordered Model Transformation

An ordered model transformation is a nested set of transformations between an ordered source model and an ordered target model, in which each transformation is governed by its own pre and postconditions, but structurally dependent on its parent. An ordered model transformation is either built from a solitary data component, or from a data link poset topped off by a data component. In terms of the ladder analogy, an ordered model transformation is a fully-formed tree of ladders, which captures the essential characteristics of every transformation between the ends of the rungs—including the root rung—and every transformation between the risers.

The following type theoretical definition replaces the informal set theoretical definition given earlier.

Definition 45. *(Ordered Model Transformation - Type)* An ordered model transformation between a source model rooted at X and a target model rooted at Y is an inhabitant of type $Tran\ X\ Y$.

Formation Rule

The $Tran$ formation rule, which is given by

$$\frac{X : U_0 \quad Y : U_0}{Tran\ X\ Y : U_1}\ (Tran\ F)\ ,$$

asserts that $Tran\ X\ Y$ is an inhabitant of U_1, if X and Y (the root classes) are inhabitants of U_0.

Introduction Rules

There are two $Tran$ introduction rules. The first rule, which is given by

$$\frac{X : U_0 \quad Y : U_0 \quad d : Data\ X\ Y}{@^1_{Tran}\ X\ Y\ d : Tran\ X\ Y}\ (Tran\ I_1)\ ,$$

asserts that $(@^1_{Tran}\ X\ Y\ d)$ is an ordered model transformation on X and Y, if d is a data component on X and Y. This kind of transformation is analogous to a one-runged ladder without risers, hardly a ladder at all one might argue! However, this *is* a special case. If t^1_{AW} is an ordered model transformation on A and W (see Figure 5.3) which just comprises the data component on A and W, then t^1_{AW} is given by

$$\frac{A : U_0 \quad W : U_0 \quad d_{AW} : Data\ A\ W}{(t^1_{AW} =_{df} @^1_{Tran}\ A\ W\ d_{AW}) : Tran\ A\ W}\ (Tran\ I_1)\ ,$$

where d_{AW} is as defined in Section 5.2.1. The second rule, which is given by

$$\frac{X : U_0 \quad Y : U_0 \quad \begin{array}{l} d : Data\ X\ Y \\ p : Poset\ X\ Y \end{array}}{@^2_{Tran}\ X\ Y\ d\ p : Tran\ X\ Y}\ (Tran\ I_2)\ ,$$

asserts that $(@^2_{Tran}\, X\, Y\, d\, p)$ is an ordered model transformation on X and Y, if d is a data component on X and Y, and p is a data link poset on X and Y. If t^2_{AW} is an ordered model transformation on A and W (again, see Figure 5.3) which comprises the entire tree of transformations rooted at A and W, then t^2_{AW} is given by

$$\frac{\begin{array}{cc} A\colon U_0 & W\colon U_0 \end{array} \qquad \begin{array}{l} d_{AW}\colon \mathit{Data}\, A\, W \\ p_{AW}\colon \mathit{Poset}\, A\, W \end{array}}{(t^2_{AW} =_{df} @^2_{Tran}\, A\, W\, d_{AW}\, p_{AW})\colon \mathit{Tran}\, A\, W}\ (\mathit{Tran}\, I_2)\ ,$$

where p_{AW} is as defined in Section 5.2.4. The step by step construction of t^2_{AW} is shown in Figures 5.9 through 5.12.

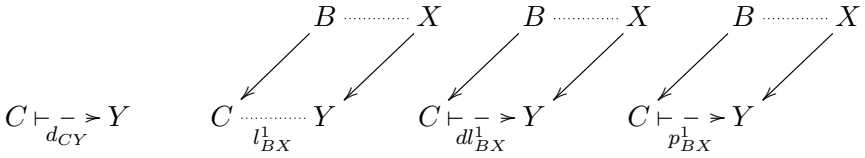

Figure 5.9: The construction of t^2_{AW} part 1 of 4. A data component d_{CY} is combined with a link component l^1_{BX} to produce a data link component dl^1_{BX}, and thence a data link poset p^1_{BX}.

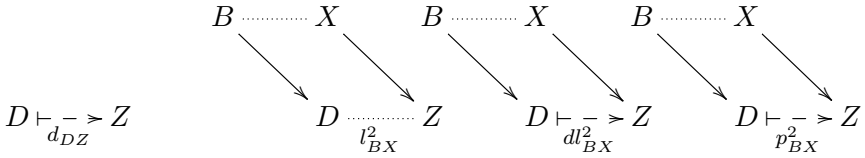

Figure 5.10: The construction of t^2_{AW} part 2 of 4. A data component d_{DZ} is combined with a link component l^2_{BX} to produce a data link component dl^2_{BX}, and thence a data link poset p^2_{BX}.

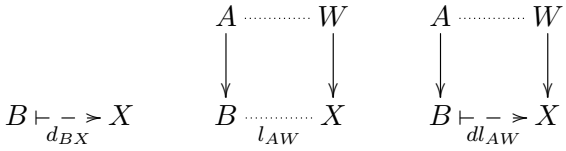

Figure 5.11: The construction of t^2_{AW} part 3 of 4. A data component d_{BX} is combined with a link component l_{AW} to produce a data link component dl_{AW}.

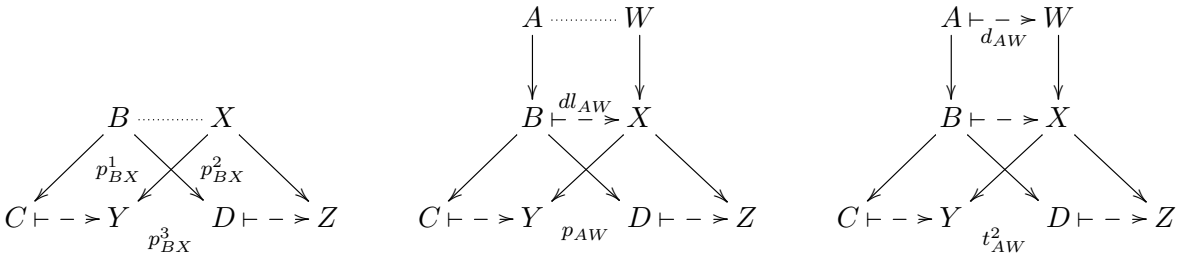

Figure 5.12: The construction of t^2_{AW} part 4 of 4. The data link posets p^1_{BX} and p^2_{BX} are merged into a new data link poset p^3_{BX}, keyed off the same stem. The data link component dl_{AW} and the data component d_{AW} round off the construction.

Elimination Rules

The $Tran$ elimination rules are given by

$$\frac{\begin{array}{l} P\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,(Tran\ X\ Y \to U_n) \\ X\colon U_0 \\ Y\colon U_0 \\ t\colon Tran\ X\ Y \\ i\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall d\colon Data\ X\ Y\,.\,P\ X\ Y\ (@^1_{Tran}\ X\ Y\ d) \\ j\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall d\colon Data\ X\ Y\,.\,\forall p\colon Poset\ X\ Y\,. \\ \qquad P\ X\ Y\ (@^2_{Tran}\ X\ Y\ d\ p) \end{array}}{@^{-1}_{Tran}\ P\ X\ Y\ t\ i\ j\colon P\ X\ Y\ t}\ (Tran\ E_n)\ ,$$

for $n = 0, 1, 2, \ldots$. Each rule asserts, for a particular universe U_n, that $P\ X\ Y\ t$ is inhabited for all ordered model transformations t, if $P\ X\ Y\ (@^1_{Tran}\ X\ Y\ d)$ is inhabited for all data components d, and $P\ X\ Y\ (@^2_{Tran}\ X\ Y\ d\ p)$ is inhabited for all data components d, and all data link posets p, i.e. if $P\ X\ Y\ t$ is inhabited every which way that t can be constructed.

Computation Rules

The $Tran$ computation rules are given by

$$@^{-1}_{Tran}\ P\ X\ Y\ (@^1_{Tran}\ X\ Y\ d)\ i\ j \twoheadrightarrow i\ X\ Y\ d$$
$$@^{-1}_{Tran}\ P\ X\ Y\ (@^2_{Tran}\ X\ Y\ d\ p)\ i\ j \twoheadrightarrow j\ X\ Y\ d\ p\ .$$

Each rule asserts that the term on the right hand side is a simplification of the term on the left hand side.

Coq Listing 44. *(Tran)* The encoding of an ordered model transformation is shown below. Note that there are two constructors: the first constructor requires a data component, and the second constructor requires a data component and a data link poset.

```
Inductive Tran : Set -> Set -> Type :=
```

```
Build_Tran_1 :
   forall X : Set,
   forall Y : Set,
   Data X Y ->
      Tran X Y |
Build_Tran_2 :
   forall X : Set,
   forall Y : Set,
   Data X Y ->
   Poset X Y ->
      Tran X Y.
```

In future, in answering the question "what is an ordered model transformation?", the answer now will not be as stated previously, i.e. "a nested set of transformations between an ordered source model and an ordered target model, and so on", which it undoubtedly still is, but simply "an inhabitant of type $Tran\ X\ Y$ for some X and Y".

5.2.7 Component Summary

For ease of reference, this section contains a summary of the component definitions associated with Figure 5.3. It also introduces two new components, a data link poset p^2_{AW}, and an ordered model transformation t^3_{AW}, both on A and W, which the author found useful in describing the logical interpretation of an ordered model transformation in the next section. See Figure 5.13.

$$\begin{array}{ccc} A & \overset{t^3_{AW}}{\vdash\!\!\!- - - - - - \succ} & W \\ {\scriptstyle R_1}\downarrow & {\scriptstyle p^2_{AW}} & \downarrow{\scriptstyle S_1} \\ B & \vdash\!\!\!- - - - - - - - \succ & X \end{array}$$

Figure 5.13: A third ordered model transformation on A and W, containing just two data components.

The component definitions are as follows.

$$\begin{array}{ll}
d_{AW} =_{df} @_{Data}\ A\ W\ Pre_A\ Post_W & dl^2_{BX} =_{df} @_{DataLink}\ B\ X\ D\ Z\ d_{DZ}\ l^2_{BX} \\
d_{BX} =_{df} @_{Data}\ B\ X\ Pre_B\ Post_X & p_{AW} =_{df} @^2_{Poset}\ A\ W\ B\ X\ dl_{AW}\ p^3_{BX} \\
d_{CY} =_{df} @_{Data}\ C\ Y\ Pre_C\ Post_Y & p^2_{AW} =_{df} @^1_{Poset}\ A\ W\ B\ X\ dl_{AW} \\
d_{DZ} =_{df} @_{Data}\ D\ Z\ Pre_D\ Post_Z & p^1_{BX} =_{df} @^1_{Poset}\ B\ X\ C\ Y\ dl^1_{BX} \\
l_{AW} =_{df} @^1_{Link}\ A\ W\ B\ X\ R_1\ S_1 & p^2_{BX} =_{df} @^1_{Poset}\ B\ X\ D\ Z\ dl^2_{BX} \\
l^1_{BX} =_{df} @^2_{Link}\ B\ X\ C\ Y\ R_2\ S_2 & p^3_{BX} =_{df} @^3_{Poset}\ B\ X\ p^1_{BX}\ p^2_{BX} \\
l^2_{BX} =_{df} @^1_{Link}\ B\ X\ D\ Z\ R_3\ S_3 & t^1_{AW} =_{df} @^1_{Tran}\ A\ W\ d_{AW} \\
dl_{AW} =_{df} @_{DataLink}\ A\ W\ B\ X\ d_{BX}\ l_{AW} & t^2_{AW} =_{df} @^2_{Tran}\ A\ W\ d_{AW}\ p_{AW} \\
dl^1_{BX} =_{df} @_{DataLink}\ B\ X\ C\ Y\ d_{CY}\ l^1_{BX} & t^3_{AW} =_{df} @^2_{Tran}\ A\ W\ d_{AW}\ p^2_{AW}
\end{array}$$

5.3 Logical Interpretation

As its type definition implies, the logical interpretation of t^1_{AW} is given by

$$\forall a\colon A \,.\, Pre_A\, a \rightarrow \exists w\colon W \,.\, Post_W\, a\, w \;,$$

i.e. the type of functions that take an object a of A, to a function that takes a proof of $Pre_A\, a$, to a pair $\langle \overline{w},\, p \rangle$, where $\overline{w}$ is an object of W, and p is a proof of $Post_W\, a\, \overline{w}$. Or to put it simply, the type of functions that transform A to W subject to Pre_A and $Post_W$.

Similarly, the logical interpretation of t^3_{AW} is given by

$$\begin{aligned}\forall a\colon A \,.\, Pre_A\, a \rightarrow \exists w\colon W \,.\, Post_W\, a\, w \,\wedge \\ \forall b\colon B \,.\, Pre_B\, b \,\wedge b = R_1\, a \rightarrow \exists x\colon X \,.\, Post_X\, b\, x \,\wedge x = S_1\, w \;,\end{aligned}$$

i.e. the type of functions that take an object a of A, to a function that takes a proof of $Pre_A\, a$, to a pair $\langle \overline{w},\, \langle p_1,\, p_2 \rangle \rangle$, where $\overline{w}$ is an object of W, p_1 is a proof of $Post_W\, a\, \overline{w}$, and p_2 is a proof of

$$\forall b\colon B \,.\, Pre_B\, b \,\wedge b = R_1\, a \rightarrow \exists x\colon X \,.\, Post_X\, b\, x \,\wedge x = S_1\, \overline{w} \;,$$

i.e. the type of functions that take an object b of B, to a function that takes a proof of $Pre_B\, b \,\wedge b = R_1\, a$, to a pair $\langle \overline{x},\, p_3 \rangle$, where $\overline{x}$ is an object of X, and p_3 is a proof of

$$Post_X\, b\, \overline{x} \,\wedge \overline{x} = S_1\, \overline{w} \;.$$

Or to put it simply, the type of functions that transform A to W subject to Pre_A and $Post_W$; B to X subject to Pre_B and $Post_X$; and R_1 (the relation between A and B) to S_1 (the relation between W and X).

Clearly, the logical interpretation of any *specific* ordered model transformation, no matter how large, could be interpreted in a similar manner. However, what about a *non-specific* ordered model transformation? What is its logical interpretation? To begin answering these questions, it is apposite to introduce a number of utility functions, to enhance the readability of later definitions.

5.3.1 Utility Functions

This section defines the following utility functions: $Data_{Pre}$, $Data_{Post}$, $Link_{Pre}$, $Link_{Post}$, $DataLink_{Data}$, and $DataLink_{Link}$. As their names suggest, these functions extract the constituent parts of data components, link components and data link components. For instance, $Data_{Pre}$ supports the notion that just as a precondition p is required to construct a data component d, so p would emerge if d were destructed.

Definition 46. *($Data_{Pre}$)* The precondition of a data component d on X and Y is given by

$$Data_{Pre}\, X\, Y\, d \;,$$

where

$$\begin{aligned} Data_{Pre} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda d\,.\,@^{-1}_{Data}\,P\,X\,Y\,d\,i \\ P &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda d\,.\,X \to U_0 \\ i &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda Pre\,.\,\lambda Post\,.\,Pre\,. \end{aligned}$$

$@^{-1}_{Data}$ is the data component destructor, as defined in Section 5.2.1.

Coq Listing 45. *($Data_{Pre}$)*

```
Definition DataPre (X : Set) (Y : Set) (d : Data X Y) : X -> Prop :=
   match d with
      Build_Data X Y Pre Post => Pre
   end.
```

Example 14. *($Data_{Pre}$)* The precondition of d_{AW} is given by

$$\begin{aligned} Data_{Pre}\,A\,W\,d_{AW} &\twoheadrightarrow @^{-1}_{Data}\,P\,A\,W\,(@_{Data}\,A\,W\,Pre_A\,Post_P)\,i \\ &\twoheadrightarrow i\,A\,W\,Pre_A\,Post_P \\ &\twoheadrightarrow Pre_A\,. \end{aligned}$$

Definition 47. *($Data_{Post}$)* The postcondition of a data component d on X and Y is given by

$$Data_{Post}\,X\,Y\,d\,,$$

where

$$\begin{aligned} Data_{Post} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda d\,.\,@^{-1}_{Data}\,P\,X\,Y\,d\,i \\ P &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda d\,.\,X \to Y \to U_0 \\ i &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda Pre\,.\,\lambda Post\,.\,Post\,. \end{aligned}$$

Coq Listing 46. *($Data_{Post}$)*

```
Definition DataPost (X : Set) (Y : Set) (d : Data X Y) : X -> Y -> Prop :=
   match d with
      Build_Data X Y Pre Post => Post
   end.
```

Definition 48. *($Link_{Pre}$)* The precondition of a link component l on X, Y, X' and Y' is given by

$$Link_{Pre}\,X\,Y\,X'\,Y'\,l\,,$$

where

$$\begin{aligned} Link_{Pre} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda l\,.\,@^{-1}_{Link}\,P\,X\,Y\,X'\,Y'\,l\,i\,j \\ P &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda l\,.\,X \to X' \to U_0 \\ i &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda r\,.\,\lambda s\,.\,(\lambda x\,.\,\lambda x'\,.\,x' = r\,x) \\ j &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda r\,.\,\lambda s\,.\,(\lambda x\,.\,\lambda x'\,.\,x' \in r\,x)\,. \end{aligned}$$

$@^{-1}_{Link}$ is the link component destructor, as defined in Section 5.2.2. The preconditions of an ordered model transformation are generally composed of two parts: one part is concerned with data, and is extracted from a data component by means of the $Data_{Pre}$ utility; the other part is concerned with linkage, and is extracted from a link component by means of the $Link_{Pre}$ utility.

Coq Listing 47. *($Link_{Pre}$)*

```
Definition LinkPre (X : Set) (Y : Set) (X' : Set) (Y' : Set)
      (l : Link X Y X' Y') : (X -> X' -> Prop) :=
   match l with
      Build_Link_1 X Y X' Y' r s =>
         fun (x : X) (x' : X') => x' = r x |
      Build_Link_2 X Y X' Y' r s =>
         fun (x : X) (x' : X') => In x' (r x)
   end.
```

Example 15. *($Link_{Pre}$)* The precondition of l_{AW} is given by

$$\begin{aligned} Link_{Pre}\, A\, W\, B\, X\, l_{AW} &\twoheadrightarrow @^{-1}_{Link}\, P\, A\, W\, B\, X\, (@^{1}_{Link}\, A\, W\, B\, X\, R_1\, S_1)\, i\, j \\ &\twoheadrightarrow i\, A\, W\, B\, X\, R_1\, S_1 \\ &\twoheadrightarrow \lambda x\,.\,\lambda x'\,.\,x' = R_1\, x\,. \end{aligned}$$

Definition 49. *($Link_{Post}$)* The postcondition of a link component l on X, Y, X' and Y' is given by

$$Link_{Post}\, X\, Y\, X'\, Y'\, l\,,$$

where

$$\begin{aligned} Link_{Post} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda l\,.\,@^{-1}_{Link}\, P\, X\, Y\, X'\, Y'\, l\, i\, j \\ P &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda l\,.\,Y \to Y' \to U_0 \\ i &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda r\,.\,\lambda s\,.\,(\lambda y\,.\,\lambda y'\,.\,y' = s\, y) \\ j &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda r\,.\,\lambda s\,.\,(\lambda y\,.\,\lambda y'\,.\,y' \in s\, y)\,. \end{aligned}$$

Coq Listing 48. *($Link_{Post}$)*

```
Definition LinkPost (X : Set) (Y : Set) (X' : Set) (Y' : Set)
      (l : Link X Y X' Y') : (Y -> Y' -> Prop) :=
   match l with
      Build_Link_1 X Y X' Y' r s =>
         fun (y : Y) (y' : Y') => y' = s y |
      Build_Link_2 X Y X' Y' r s =>
         fun (y : Y) (y' : Y') => In y' (s y)
   end.
```

Definition 50. *($DataLink_{Data}$)* The data component of a data link component dl on X, Y, X' and Y' is given by

$$DataLink_{Data}\, X\, Y\, X'\, Y'\, dl\,,$$

where

$$\begin{aligned} DataLink_{Data} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda dl\,.\,@^{-1}_{DataLink}\,P\,X\,Y\,X'\,Y'\,dl\,i \\ P &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda dl\,.\,Data\,X'\,Y' \\ i &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda d'\,.\,\lambda l\,.\,d'\,. \end{aligned}$$

$@^{-1}_{DataLink}$ is the data link component destructor, as defined in Section 5.2.3.

Coq Listing 49. *($DataLink_{Data}$)*

```
Definition DataLinkData (X : Set) (Y : Set) (X' : Set) (Y' : Set)
      (dl : DataLink X Y X' Y') : Data X' Y' :=
   match dl with
      Build_DataLink X Y X' Y' d' l => d'
   end.
```

Example 16. *($DataLink_{Data}$)* The data component of dl_{AW} is given by

$$\begin{aligned} DataLink_{Data}\,A\,W\,B\,X\,dl_{AW} &\twoheadrightarrow @^{-1}_{DataLink}\,P\,A\,W\,B\,X\,(@_{DataLink}\,A\,W\,B\,X\,d_{BX}\,l_{AW})\,i \\ &\twoheadrightarrow i\,A\,W\,B\,X\,d_{BX}\,l_{AW} \\ &\twoheadrightarrow d_{BX}\,. \end{aligned}$$

Definition 51. *($DataLink_{Link}$)* The link component of a data link component dl on X, Y, X' and Y' is given by

$$DataLink_{Link}\,X\,Y\,X'\,Y'\,dl\,,$$

where

$$\begin{aligned} DataLink_{Data} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda dl\,.\,@^{-1}_{DataLink}\,P\,X\,Y\,X'\,Y'\,dl\,i \\ P &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda dl\,.\,Link\,X\,Y\,X'\,Y' \\ i &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda d'\,.\,\lambda l\,.\,l\,. \end{aligned}$$

Coq Listing 50. *($DataLink_{Link}$)*

```
Definition DataLinkLink (X : Set) (Y : Set) (X' : Set) (Y' : Set)
      (dl : DataLink X Y X' Y') : Link X Y X' Y' :=
   match dl with
      Build_DataLink X Y X' Y' d' l => l
   end.
```

With the utility functions in place, it is now time to formally define the logical interpretation of a *non-specific* ordered model transformation, that is to say a function Log_{Tran} that takes a non-specific ordered model transformation to its logical interpretation.

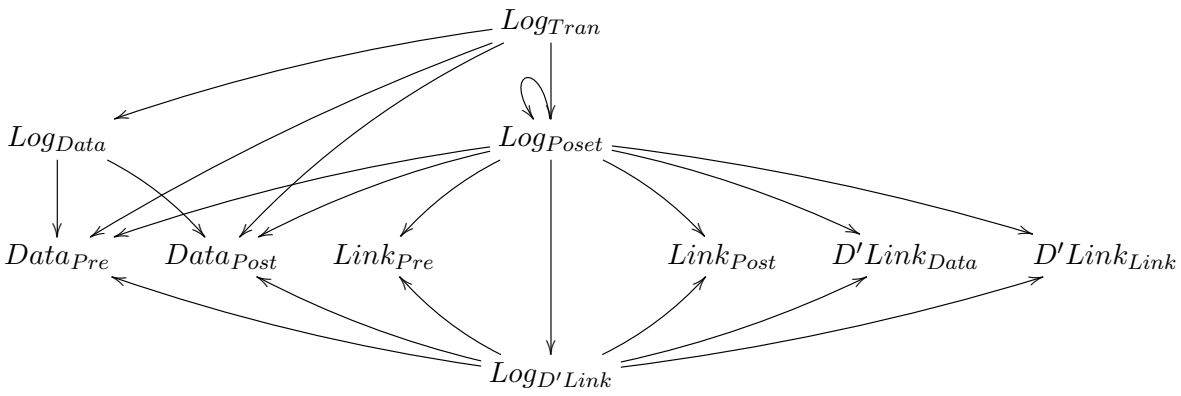

Figure 5.14: The call tree of Log_{Tran}.

5.3.2 *Log* Functions

The call tree of Log_{Tran} is shown in Figure 5.14. Clearly, Log_{Tran} delegates part of its responsibility to Log_{Data} (a function that takes a data component to its logical interpretation), and part to Log_{Poset} (a recursive function that takes a data link poset to its logical interpretation), which in turns delegates part of its responsibility to $Log_{DataLink}$ (a function that takes a data link component to its logical interpretation). The utility functions $Data_{Pre}$, $Data_{Post}$ and so on, round off the call tree.

The *Log* functions, together with appropriate examples, are defined as follows.

Definition 52. *(Log_{Data})* The logical interpretation of a data component d on X and Y is given by

$$Log_{Data}\ X\ Y\ d\ ,$$

where

$$Log_{Data} =_{df} \lambda X\,.\,\lambda Y\,.\,\lambda d\,.\,\forall x\colon X\,.\,(Data_{Pre}\ X\ Y\ d)\ x \to \exists y\colon Y\,.\,(Data_{Post}\ X\ Y\ d)\ x\ y\ .$$

Coq Listing 51. *(Log_{Data})*

```
Definition LogData (X : Set) (Y : Set) (d : Data X Y) : Prop :=
   forall x : X, (DataPre X Y d) x ->
      exists y : Y, (DataPost X Y d) x y.
```

Definition 53. *(Log_{Link})* The logical interpretation of a link component l on X, Y, X', and Y' is given by

$$Log_{Link}\ X\ Y\ X'\ Y'\ l\ ,$$

where

$$\begin{aligned} Log_{Link} =_{df}\ & \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda l\,. \\ & \lambda x\,.\,\lambda y\,.\,\forall x'\colon X'\,.\,(Link_{Pre}\ X\ Y\ X'\ Y'\ l)\ x\ x' \to \\ & \exists y'\colon Y'\,.\,(Link_{Post}\ X\ Y\ X'\ Y'\ l)\ y\ y'\ . \end{aligned}$$

This function is not part of the call tree of Log_{Tran}. However, it is utilised in the specification of Theorem 3 in Section 6.1.

Coq Listing 52. *(Log_{Link})*

```
Definition LogLink (X : Set) (Y : Set) (X' : Set) (Y' : Set)
   (l : Link X Y X' Y') : (X -> Y -> Prop) :=

   fun (x : X) (y : Y) =>
      forall x' : X', (LinkPre X Y X' Y' l) x x' ->
         exists y' : Y', (LinkPost X Y X' Y' l) y y'.
```

Example 17. *(Log_{Link})* The logical interpretation of l_{AW} is given by

$$\begin{aligned} Log_{Link}\, A\, W\, B\, X\, l_{AW} &\twoheadrightarrow \lambda x\,.\,\lambda y\,.\,\forall x'\colon B\,.\,(Link_{Pre}\, A\, W\, B\, X\, l_{AW})\, x\, x' \rightarrow \\ &\qquad \exists y'\colon X\,.\,(Link_{Post}\, A\, W\, B\, X\, l_{AW})\, y\, y' \\ &\twoheadrightarrow \lambda x\,.\,\lambda y\,.\,\forall x'\colon B\,.\,(\lambda x\,.\,\lambda x'\,.\,x' = R_1\, x)\, x\, x' \rightarrow \\ &\qquad \exists y'\colon X\,.\,(\lambda y\,.\,\lambda y'\,.\,y' = S_1\, y)\, y\, y' \\ &\twoheadrightarrow \lambda x\,.\,\lambda y\,.\,\forall x'\colon B\,.\,x' = R_1\, x \rightarrow \exists y'\colon X\,.\,y' = S_1\, y \\ &\rightarrow_{\alpha} \lambda a\,.\,\lambda w\,.\,\forall b\colon B\,.\,b = R_1\, a \rightarrow \exists x\colon X\,.\,x = S_1\, w\,. \end{aligned}$$

Definition 54. *($Log_{DataLink}$)* The logical interpretation of a data link component dl on X, Y, X', and Y' is given by

$$Log_{DataLink}\, X\, Y\, X'\, Y'\, dl\,,$$

where

$$\begin{aligned} &Log_{DataLink} =_{df} \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda dl\,. \\ &\quad \lambda x\,.\,\lambda y\,. \\ &\qquad \forall x'\colon X'\,.\,(Data_{Pre}\, X'\, Y'\, (DataLink_{Data}\, X\, Y\, X'\, Y'\, dl))\, x' \wedge \\ &\qquad\quad (Link_{Pre}\, X\, Y\, X'\, Y'\, (DataLink_{Link}\, X\, Y\, X'\, Y'\, dl))\, x\, x' \rightarrow \\ &\qquad\qquad \exists y'\colon Y'\,.\,(Data_{Post}\, X'\, Y'\, (DataLink_{Data}\, X\, Y\, X'\, Y'\, dl))\, x'\, y' \wedge \\ &\qquad\qquad\quad (Link_{Post}\, X\, Y\, X'\, Y'\, (DataLink_{Link}\, X\, Y\, X'\, Y'\, dl))\, y\, y'\,. \end{aligned}$$

Coq Listing 53. *($Log_{DataLink}$)*

```
Definition LogDataLink (X : Set) (Y : Set) (X' : Set) (Y' : Set)
      (dl : DataLink X Y X' Y') : (X -> Y -> Prop) :=
   fun (x : X) (y : Y) =>
      forall x' : X', (DataPre X' Y' (DataLinkData X Y X' Y' dl)) x' /\
         (LinkPre X Y X' Y' (DataLinkLink X Y X' Y' dl)) x x' ->
            exists y' : Y', (DataPost X' Y' (DataLinkData X Y X' Y' dl)) x' y' /\
               (LinkPost X Y X' Y' (DataLinkLink X Y X' Y' dl)) y y'.
```

Example 18. *($Log_{DataLink}$)* The logical interpretation of dl_{AW} is given by

$$
\begin{array}{l}
Log_{DataLink}\ A\ W\ B\ X\ dl_{AW} \\
\twoheadrightarrow \lambda x\,.\,\lambda y\,. \\
\quad \forall x'\colon B\,.\,(Data_{Pre}\ B\ X\ (DataLink_{Data}\ A\ W\ B\ X\ dl_{AW}))\ x' \wedge \\
\quad\quad (Link_{Pre}\ A\ W\ B\ X\ (DataLink_{Link}\ A\ W\ B\ X\ dl_{AW}))\ x\ x' \rightarrow \\
\quad\quad\quad \exists y'\colon X\,.\,(Data_{Post}\ B\ X\ (DataLink_{Data}\ A\ W\ B\ X\ dl_{AW}))\ x'\ y' \wedge \\
\quad\quad\quad\quad (Link_{Post}\ A\ W\ B\ X\ (DataLink_{Link}\ A\ W\ B\ X\ dl_{AW}))\ y\ y' \\
\twoheadrightarrow \lambda x\,.\,\lambda y\,. \\
\quad \forall x'\colon B\,.\,(Data_{Pre}\ B\ X\ d_{BX})\ x' \wedge \\
\quad\quad (Link_{Pre}\ A\ W\ B\ X\ l_{AW})\ x\ x' \rightarrow \\
\quad\quad\quad \exists y'\colon X\,.\,(Data_{Post}\ B\ X\ d_{BX})\ x'\ y' \wedge \\
\quad\quad\quad\quad (Link_{Post}\ A\ W\ B\ X\ l_{AW})\ y\ y' \\
\twoheadrightarrow \lambda x\,.\,\lambda y\,. \\
\quad \forall x'\colon B\,.\,Pre_B\ x' \wedge \\
\quad\quad (\lambda x\,.\,\lambda x'\,.\,x' = R_1\ x)\ x\ x' \rightarrow \\
\quad\quad\quad \exists y'\colon X\,.\,Post_X\ x'\ y' \wedge \\
\quad\quad\quad\quad (\lambda y\,.\,\lambda y'\,.\,y' = S_1\ y)\ y\ y' \\
\twoheadrightarrow \lambda x\,.\,\lambda y\,.\,\forall x'\colon B\,.\,Pre_B\ x'\ \wedge x' = R_1\ x \rightarrow \exists y'\colon X\,.\,Post_X\ x'\ y'\ \wedge y' = S_1\ y \\
\rightarrow_\alpha \lambda a\,.\,\lambda w\,.\,\forall b\colon B\,.\,Pre_B\ b\ \wedge b = R_1\ a \rightarrow \exists x\colon X\,.\,Post_X\ b\ x\ \wedge x = S_1\ w\,.
\end{array}
$$

Definition 55. *(Log_{Poset})* The logical interpretation of a data link poset p on X and Y is given by

$$Log_{Poset}\ X\ Y\ p\ ,$$

where

$$
\begin{array}{rcl}
Log_{Poset} & =_{df} & \lambda X\,.\,\lambda Y\,.\,\lambda p\,.\,@^{-1}_{Poset}\ P\ X\ Y\ p\ i\ j\ k \\
P & =_{df} & \lambda X\,.\,\lambda Y\,.\,\lambda p\,.\,X \rightarrow Y \rightarrow U_0 \\
i & =_{df} & \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda dl\,.\,Log_{DataLink}\ X\ Y\ X'\ Y'\ dl \\
j & =_{df} & \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda dl\,.\,\lambda p'\,. \\
 & & \quad \lambda x\,.\,\lambda y\,. \\
 & & \quad\quad \forall x'\colon X'\,.\,(Data_{Pre}\ X'\ Y'\ (DataLink_{Data}\ X\ Y\ X'\ Y'\ dl))\ x' \wedge \\
 & & \quad\quad\quad (Link_{Pre}\ X\ Y\ X'\ Y'\ (DataLink_{Link}\ X\ Y\ X'\ Y'\ dl))\ x\ x' \rightarrow \\
 & & \quad\quad\quad\quad \exists y'\colon Y'\,.\,(Data_{Post}\ X'\ Y'\ (DataLink_{Data}\ X\ Y\ X'\ Y'\ dl))\ x'\ y' \wedge \\
 & & \quad\quad\quad\quad\quad (Link_{Post}\ X\ Y\ X'\ Y'\ (DataLink_{Link}\ X\ Y\ X'\ Y'\ dl))\ y\ y' \wedge \\
 & & \quad\quad\quad\quad\quad\quad Log_{Poset}\ X'\ Y'\ p'\ x'\ y' \\
k & =_{df} & \lambda X\,.\,\lambda Y\,.\,\lambda p_1\,.\,\lambda p_2\,. \\
 & & \quad \lambda x\,.\,\lambda y\,.\,Log_{Poset}\ X\ Y\ p_1\ x\ y\ \wedge Log_{Poset}\ X\ Y\ p_2\ x\ y\ .
\end{array}
$$

Coq Listing 54. *(Log_{Poset})*

```
Fixpoint LogPoset (X : Set) (Y : Set) (p : Poset X Y) : X -> Y -> Prop :=
   match p in Poset X Y return X -> Y -> Prop with
      Build_Poset_1 X Y X' Y' dl  =>
         LogDataLink X Y X' Y' dl |
      Build_Poset_2 X Y X' Y' dl p' =>
         fun (x : X) (y : Y) =>
            forall x' : X', (DataPre X' Y' (DataLinkData X Y X' Y' dl)) x' /\
               (LinkPre X Y X' Y' (DataLinkLink X Y X' Y' dl)) x x' ->
                  exists y' : Y', (DataPost X' Y' (DataLinkData X Y X' Y' dl)) x' y' /\
                     (LinkPost X Y X' Y' (DataLinkLink X Y X' Y' dl)) y y' /\
                           LogPoset X' Y' p' x' y' |
      Build_Poset_3 X Y p1 p2 =>
         fun (x : X) (y : Y) =>
            LogPoset X Y p1 x y /\ LogPoset X Y p2 x y
   end.
```

Example 19. *(Log_{Poset})* The logical interpretation of p_{AW} is given by

$$\begin{aligned} Log_{Poset}\, A\, W\, p_{AW} &\twoheadrightarrow @^{-1}_{Poset}\, P\, A\, W\, (@^{2}_{Poset}\, A\, W\, B\, X\, dl_{AW}\, p^{3}_{BX})\, i\, j\, k \\ &\twoheadrightarrow j\, A\, W\, B\, X\, dl_{AW}\, p^{3}_{BX}\, (@^{-1}_{Poset}\, P\, B\, X\, p^{3}_{BX}\, i\, j\, k)\,. \end{aligned}$$

Now

$$\begin{aligned} @^{-1}_{Poset}\, P\, B\, X\, p^{3}_{BX}\, i\, j\, k &\twoheadrightarrow @^{-1}_{Poset}\, P\, B\, X\, (@^{3}_{Poset}\, B\, X\, p^{1}_{BX}\, p^{2}_{BX})\, i\, j\, k \\ &\twoheadrightarrow k\, B\, X\, p^{1}_{BX}\, p^{2}_{BX}\, (@^{-1}_{Poset}\, P\, B\, X\, p^{1}_{BX}\, i\, j\, k)\, (@^{-1}_{Poset}\, P\, B\, X\, p^{2}_{BX}\, i\, j\, k)\,; \end{aligned}$$

and

$$\begin{aligned} @^{-1}_{Poset}\, P\, B\, X\, p^{1}_{BX}\, i\, j\, k &\twoheadrightarrow @^{-1}_{Poset}\, P\, B\, X\, (@^{1}_{Poset}\, B\, X\, C\, Y\, dl^{1}_{BX})\, i\, j\, k \\ &\twoheadrightarrow i\, B\, X\, C\, Y\, dl^{1}_{BX} \\ &\twoheadrightarrow Log_{DataLink}\, B\, X\, C\, Y\, dl^{1}_{BX}\,; \end{aligned}$$

and

$$\begin{aligned} @^{-1}_{Poset}\, P\, B\, X\, p^{2}_{BX}\, i\, j\, k &\twoheadrightarrow @^{-1}_{Poset}\, P\, B\, X\, (@^{1}_{Poset}\, B\, X\, D\, Z\, dl^{2}_{BX})\, i\, j\, k \\ &\twoheadrightarrow i\, B\, X\, D\, Z\, dl^{2}_{BX} \\ &\twoheadrightarrow Log_{DataLink}\, B\, X\, D\, Z\, dl^{2}_{BX}\,; \end{aligned}$$

and

$$\begin{aligned} &Log_{DataLink}\, B\, X\, C\, Y\, dl^{1}_{BX} \\ &\twoheadrightarrow \lambda b\,.\,\lambda x\,.\,\forall c\colon C\,.\,Pre_C\, c\, \wedge c \in R_2\, b \to \exists y\colon Y\,.\,Post_Y\, c\, y\, \wedge y \in S_2\, x\,; \end{aligned}$$

and

$$Log_{DataLink}\, B\, X\, D\, Z\, dl^{2}_{BX}$$

$$\twoheadrightarrow \lambda b\,.\,\lambda x\,.\,\forall d\colon D\,.\,Pre_D\ d\ \wedge d = R_3\ b \rightarrow \exists z\colon Z\,.\,Post_Z\ d\ z\ \wedge z = S_3\ x\ .$$

$\therefore$

$$\begin{aligned}
&@^{-1}_{Poset}\ P\ B\ X\ p^3_{BX}\ i\ j\ k\\
&\twoheadrightarrow \lambda b\,.\,\lambda x\,.\,(Log_{DataLink}\ B\ X\ C\ Y\ dl^1_{BX})\ b\ x\ \wedge (Log_{DataLink}\ B\ X\ D\ Z\ dl^2_{BX})\ b\ x\\
&\twoheadrightarrow \lambda b\,.\,\lambda x\,.\\
&\qquad (\forall c\colon C\,.\,Pre_C\ c\ \wedge c \in R_2\ b \rightarrow \exists y\colon Y\,.\,Post_Y\ c\ y\ \wedge y \in S_2\ x) \wedge\\
&\qquad (\forall d\colon D\,.\,Pre_D\ d\ \wedge d = R_3\ b \rightarrow \exists z\colon Z\,.\,Post_Z\ d\ z\ \wedge z = S_3\ x)\ ;
\end{aligned}$$

and finally

$$\begin{aligned}
&Log_{Poset}\ A\ W\ p_{AW}\\
&\twoheadrightarrow \lambda a\,.\,\lambda w\,.\\
&\qquad \forall b\colon B\,.\,Pre_B\ b\ \wedge b = R_1\ a \rightarrow \exists x\colon X\,.\,Post_X\ b\ x\ \wedge x = S_1\ w\ \wedge\\
&\qquad\quad Log_{Poset}\ B\ X\ p^3_{BX}\ b\ x\\
&\twoheadrightarrow \lambda a\,.\,\lambda w\,.\\
&\qquad \forall b\colon B\,.\,Pre_B\ b\ \wedge b = R_1\ a \rightarrow \exists x\colon X\,.\,Post_X\ b\ x\ \wedge x = S_1\ w\ \wedge\\
&\qquad\quad (@^{-1}_{Poset}\ P\ B\ X\ p^3_{BX}\ i\ j\ k)\ b\ x\\
&\twoheadrightarrow \lambda a\,.\,\lambda w\,.\\
&\qquad \forall b\colon B\,.\,Pre_B\ b\ \wedge b = R_1\ a \rightarrow \exists x\colon X\,.\,Post_X\ b\ x\ \wedge x = S_1\ w\ \wedge\\
&\qquad\quad (\forall c\colon C\,.\,Pre_C\ c\ \wedge c \in R_2\ b \rightarrow \exists y\colon Y\,.\,Post_Y\ c\ y\ \wedge y \in S_2\ x) \wedge\\
&\qquad\quad (\forall d\colon D\,.\,Pre_D\ d\ \wedge d = R_3\ b \rightarrow \exists z\colon Z\,.\,Post_Z\ d\ z\ \wedge z = S_3\ x)\ .
\end{aligned}$$

Definition 56. *(Log_{Tran})* The logical interpretation of an ordered model transformation t on X and Y is given by

$$Log_{Tran}\ X\ Y\ t\ ,$$

where

$$\begin{aligned}
Log_{Tran} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda t\,.\,@^{-1}_{Tran}\ P\ X\ Y\ t\ i\ j\\
P &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda t\,.\,U_0\\
i &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda d\,.\,Log_{Data}\ X\ Y\ d\\
j &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda d\,.\,\lambda p\,.\\
&\qquad \forall x\colon X\,.\,(Data_{Pre}\ X\ Y\ d)\ x \rightarrow \exists y\colon Y\,.\,(Data_{Post}\ X\ Y\ d)\ x\ y\ \wedge\\
&\qquad\quad Log_{Poset}\ X\ Y\ p\ x\ y\ .
\end{aligned}$$

Coq Listing 55. *(Log_{Tran})*

```
Definition LogTran (X : Set) (Y : Set) (t : Tran X Y) : Prop :=
   match t with |
      Build_Tran_1 X Y d =>
```

```
      LogData X Y d |
   Build_Tran_2 X Y d p =>
      forall x : X, (DataPre X Y d) x ->
         exists y : Y, (DataPost X Y d) x y /\ LogPoset X Y p x y
end.
```

Example 20. *(Log_{Tran})* The logical interpretation of t^1_{AW} is given by

$$\begin{aligned} Log_{Tran}\, A\, W\, t^1_{AW} &\twoheadrightarrow @^{-1}_{Tran}\, P\, A\, W\, (@^1_{Tran}\, A\, W\, d_{AW})\, i\, j \\ &\twoheadrightarrow i\, X\, Y\, d_{AW} \\ &\twoheadrightarrow Log_{Data}\, X\, Y\, d_{AW} \\ &\twoheadrightarrow \forall a\colon A\,.\, Pre_A\, a \to \exists w\colon W\,.\, Post_P\, a\, w\ . \end{aligned}$$

Example 21. *(Log_{Tran})* The logical interpretation of t^2_{AW} is given by

$$\begin{aligned} &Log_{Tran}\, A\, W\, t^2_{AW} \\ &\twoheadrightarrow @^{-1}_{Tran}\, P\, A\, W\, (@^2_{Tran}\, A\, W\, d_{AW}\, p_{AW})\, i\, j \\ &\twoheadrightarrow j\, A\, W\, d_{AW}\, p_{AW} \\ &\twoheadrightarrow \forall a\colon A\,.\, (Data_{Pre}\, A\, W\, d_{AW})\, a \to \exists w\colon W\,.\, (Data_{Post}\, A\, W\, d_{AW})\, a\, w \wedge \\ &\qquad Log_{Poset}\, A\, W\, p_{AW}\, a\, w \\ &\twoheadrightarrow \forall a\colon A\,.\, Pre_A\, a \to \exists w\colon W\,.\, Post_W\, a\, w \wedge \\ &\qquad \forall b\colon B\,.\, Pre_B\, b\ \wedge b = R_1\, a \to \exists x\colon X\,.\, Post_X\, b\, x\ \wedge x = S_1\, w \wedge \\ &\qquad\quad (\forall c\colon C\,.\, Pre_C\, c\ \wedge c \in R_2\, b \to \exists y\colon Y\,.\, Post_Y\, c\, y\ \wedge y \in S_2\, x) \wedge \\ &\qquad\quad (\forall d\colon D\,.\, Pre_D\, d\ \wedge d = R_3\, b \to \exists z\colon Z\,.\, Post_Z\, d\, z\ \wedge z = S_3\, x)\ . \end{aligned}$$

5.4 Concrete Example

Figure 5.15 shows an ordered model transformation $t_{ModelSchema}$ between a simple metamodel of the Unified Modelling Language (UML), and a simple metamodel of the Structured Query Language (SQL), in which

- each model m is transformed into a schema s with the same identity as m;
- each class c in m is transformed into a table t in s with the same identity as c, if the identity of c is a positive number;
- each attribute a in c is transformed into a column o in t with the same identity as a, if the identity of a is a positive number.

$t_{ModelSchema}$ is encoded as an inhabitant of type $Tran\, Model\, Schema$ as follows.

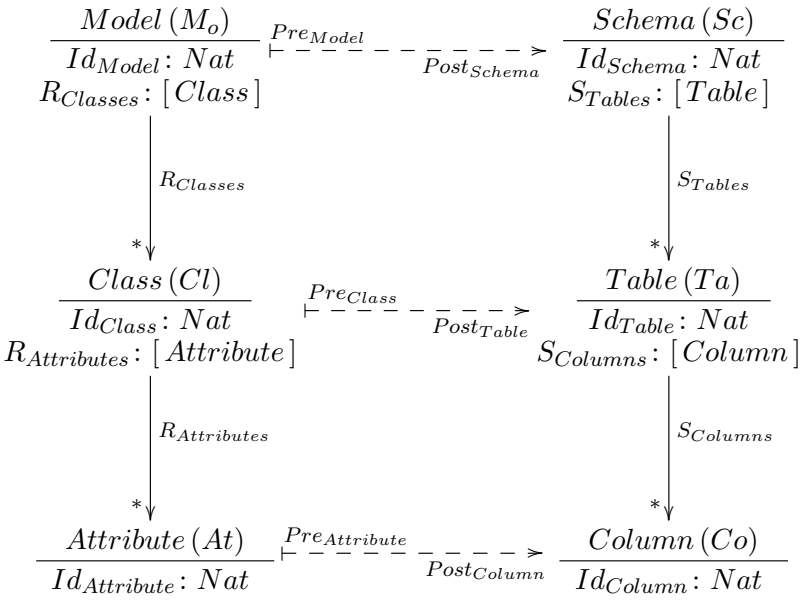

Figure 5.15: An ordered model transformation between UML and SQL.

5.4.1 Classes

The $Model$ formation, introduction, elimination and computation rules are given by

$$\frac{}{Model : U_0}\ (Model\,F)$$

$$\frac{id : Nat \quad r : [Class]}{@_{Model}\, id\, r : Model}\ (Model\,I)$$

$$\frac{\begin{array}{l} P : Model \to U_n \\ m : Model \\ i : \forall id : Nat\,.\,\forall r : [Class]\,.\,P\,(@_{Model}\, id\, r) \end{array}}{@^{-1}_{Model}\, P\, m\, i : P\, m}\ (Model\,E_n)$$

$$@^{-1}_{Model}\, P\, (@_{Model}\, id\, r) \twoheadrightarrow i\, id\, r\ .$$

The other classes are encoded in a similar manner.

5.4.2 Attributes

The Id_{Model} attribute is given by

$$(Id_{Model} =_{df} \lambda m\,.\,@^{-1}_{Model}\, P\, m\, i) : Model \to Nat\ ,$$

where

$$\begin{aligned} P &=_{df} \lambda m\,.\,Nat \\ i &=_{df} \lambda id\,.\,\lambda r\,.\,id\,. \end{aligned}$$

The other attributes are encoded in a similiar manner.

5.4.3 Preconditions

The precondition Pre_{Model} is given by

$$(Pre_{Model} =_{df} \lambda m\,.\,\top)\colon Model \to U_0\,.$$

The other preconditions are encoded in a similar manner.

5.4.4 Postconditions

The postcondition $Post_{Schema}$ is given by

$$(Post_{Schema} =_{df} \lambda m\,.\,\lambda s\,.\,Id_{Model}\,m = Id_{Schema}\,s)\colon Model \to Schema \to U_0\,.$$

The other postconditions are encoded in a similar manner.

5.4.5 Components

Finally, $t_{ModelSchema}$ is given by

$$@^2_{Tran}\,Model\;Schema\;d_{ModelSchema}\;p_{ModelSchema}\;,$$

where

$$\begin{aligned} d_{ModelSchema} &=_{df} @_{Data}\,Model\;Schema\;Pre_{Model}\;Post_{Schema} \\ d_{ClassTable} &=_{df} @_{Data}\,Class\;Table\;Pre_{Class}\;Post_{Table} \\ d_{AttributeColumn} &=_{df} @_{Data}\,Attribute\;Column\;Pre_{Attribute}\;Post_{Column} \\ l_{ModelSchema} &=_{df} @^2_{Link}\,Model\;Schema\;Class\;Table\;R_{Classes}\;S_{Tables} \\ l_{ClassTable} &=_{df} @^2_{Link}\,Class\;Table\;Attribute\;Column\;R_{Attributes}\;S_{Columns} \\ dl_{ModelSchema} &=_{df} @_{DataLink}\,Model\;Schema\;Class\;Table\;d_{ClassTable}\;l_{ModelSchema} \\ dl_{ClassTable} &=_{df} @_{DataLink}\,Class\;Table\;Attribute\;Column\;d_{AttributeColumn}\;l_{ClassTable} \\ p_{ClassTable} &=_{df} @^1_{Poset}\,Class\;Table\;Attribute\;Column\;dl_{ClassTable} \\ p_{ModelSchema} &=_{df} @^2_{Poset}\,Model\;Schema\;Class\;Table\;dl_{ModelSchema}\;p_{ClassTable}\;. \end{aligned}$$

Coq Listing 56. *($t_{ModelSchema}$)* The encoding of $t_{ModelSchema}$ is as follows.

```
Record Attribute : Set :=
   {Id_Attribute : nat}.

Record Class : Set :=
   {Id_Class : nat; R_Attributes : list Attribute}.

Record Model : Set :=
   {Id_Model : nat; R_Classes : list Class}.

Record Column : Set :=
   {Id_Column : nat}.

Record Table : Set :=
   {Id_Table : nat; S_Columns : list Column}.

Record Schema : Set :=
   {Id_Schema : nat; S_Tables : list Table}.

Definition Pre_Model (m : Model) : Prop :=
   True.

Definition Pre_Class (c : Class) : Prop :=
   Id_Class c > 0.

Definition Pre_Attribute (a : Attribute) : Prop :=
   Id_Attribute a > 0.

Definition Post_Schema (m : Model) (s : Schema) : Prop :=
   Id_Schema s = Id_Model m.

Definition Post_Table (c : Class) (t : Table) : Prop :=
   Id_Table t = Id_Class c /\ Id_Table t > 0.

Definition Post_Column (a : Attribute) (o : Column) :=
   Id_Column o = Id_Attribute a /\ Id_Column o > 0.

Definition d_Model_Schema : Data Model Schema :=
   Build_Data Model Schema Pre_Model Post_Schema.

Definition d_Class_Table : Data Class Table :=
   Build_Data Class Table Pre_Class Post_Table.

Definition d_Attribute_Column : Data Attribute Column :=
   Build_Data Attribute Column Pre_Attribute Post_Column.

Definition l_Model_Schema : Link Model Schema Class Table :=
   Build_Link_2 Model Schema Class Table R_Classes S_Tables.

Definition l_Class_Table : Link Class Table Attribute Column :=
```

```
   Build_Link_2 Class Table Attribute Column R_Attributes S_Columns.

Definition dl_Model_Schema : DataLink Model Schema Class Table :=
   Build_DataLink Model Schema Class Table d_Class_Table l_Model_Schema.

Definition dl_Class_Table : DataLink Class Table Attribute Column :=
   Build_DataLink Class Table Attribute Column d_Attribute_Column l_Class_Table.

Definition p_Class_Table : Poset Class Table :=
   Build_Poset_1 Class Table Attribute Column dl_Class_Table.

Definition p_Model_Schema : Poset Model Schema :=
   Build_Poset_2 Model Schema Class Table dl_Model_Schema p_Class_Table.

Definition t_Model_Schema : Tran Model Schema :=
   Build_Tran_2 Model Schema d_Model_Schema p_Model_Schema.
```

5.4.6 Logical Interpretation

The logical interpretation of $t_{ModelSchema}$ is given by

$$Log_{Tran}\, Model\, Schema\, t_{ModelSchema}\ ,$$

which reduces to

$$\begin{aligned}
&\forall m\colon Model\,.\,Pre_{Model}\, m \rightarrow \exists s\colon Schema\,.\,Post_{Schema}\, m\, s \wedge \\
&\quad \forall c\colon Class\,.\,Pre_{Class}\, c \wedge c \in R_{Classes}\, m \rightarrow \exists t\colon Table\,.\,Post_{Table}\, c\, t \wedge t \in S_{Tables}\, s \wedge \\
&\qquad \forall a\colon Attribute\,.\,Pre_{Attribute}\, a \wedge a \in R_{Attributes}\, c \rightarrow \\
&\qquad\quad \exists o\colon Column\,.\,Post_{Column}\, a\, o \wedge o \in S_{Columns}\, t\ .
\end{aligned}$$

Coq Listing 57. *(Logical Interpretation of $t_{ModelSchema}$)* Compare and contrast the logical interpretation of $t_{ModelSchema}$ produced below by Coq, with the hand written one above. In doing so, note that Coq replaces definiendums by their definiens, and reduces terms to normal form, during the course of an evaluation. For example, in the second line below, `True` (Coq's representation of $\top$) was derived from $Pre_{Model}\, m$.

```
Eval compute in (LogTran Model Schema t_Model_Schema).

     = forall x : Model,
       True ->
       exists y : Schema,
         (let (Id_Schema, _) := y in Id_Schema) =
         (let (Id_Model, _) := x in Id_Model) /\
         (forall x' : Class,
          1 <= (let (Id_Class, _) := x' in Id_Class) /\
          (fix In (a : Class) (l : list Class) {struct l} : Prop :=
             match l with
             | nil => False
```

```
          | b :: m => b = a \/ In a m
          end) x' (let (_, R_Classes) := x in R_Classes) ->
        exists y' : Table,
          ((let (Id_Table, _) := y' in Id_Table) =
           (let (Id_Class, _) := x' in Id_Class) /\
           1 <= (let (Id_Table, _) := y' in Id_Table)) /\
          (fix In (a : Table) (l : list Table) {struct l} : Prop :=
             match l with
             | nil => False
             | b :: m => b = a \/ In a m
             end) y' (let (_, R_Tables) := y in R_Tables) /\
          (forall x'0 : Attribute,
           1 <= (let (Id_Attribute) := x'0 in Id_Attribute) /\
           (fix In (a : Attribute) (l : list Attribute) {struct l} :
              Prop :=
              match l with
              | nil => False
              | b :: m => b = a \/ In a m
              end) x'0 (let (_, R_Attributes) := x' in R_Attributes) ->
           exists y'0 : Column,
             ((let (Id_Column) := y'0 in Id_Column) =
              (let (Id_Attribute) := x'0 in Id_Attribute) /\
              1 <= (let (Id_Column) := y'0 in Id_Column)) /\
             (fix In (a : Column) (l : list Column) {struct l} : Prop :=
                match l with
                | nil => False
                | b :: m => b = a \/ In a m
                end) y'0 (let (_, R_Columns) := y' in R_Columns)))
    : Prop
```

5.4.7 Certified Program

This section contains a proof of the logical interpretation of $t_{ModelSchema}$. In many respects, it is no different from any other proof in this thesis, apart from the fact that it deals with concrete artefacts rather than abstract ones. The proof is, in fact, part one of a two part example designed to demonstrate the difference between a monolithic proof (as given below) and a compositional proof (as given in Section 6.4). In the monolithic proof below, steps 5, 6, 8, 9 and 11 are the fundamental building blocks of the proof, whereas steps 7, 10 and 12 comprise the "glue" that binds them together. In addition to deriving the logical interpretation of the transformation by hand, Coq is enlisted to compute the certified program that implements it.

Theorem 1. The logical interpretation of $t_{ModelSchema}$ is provable, i.e.

$$\vdash Log_{Tran}\, Model\, Schema\, t_{ModelSchema}\ . \tag{5.1}$$

Proof. There are 12 steps. The reader is advised to work backwards.

1. Let

$$\begin{aligned}\Delta_1 &=_{df} [m\colon Model,\, h_1\colon Pre_{Model}\, m]\\ \Delta_2 &=_{df} \Delta_1, [c\colon Class,\, h_2\colon Pre_{Class}\, c \wedge c \in R_{Classes}\, m]\\ \Delta_3 &=_{df} \Delta_2, [a\colon Attribute,\, h_3\colon Pre_{Attribute}\, a \wedge a \in R_{Attributes}\, c]\,.\end{aligned}$$

2. Let

$$f_{Attributes} =_{df} \lambda l\,.\,@^{-1}_{[Attribute]}\, P\, i\, j\, l\colon [Attribute] \to [Column]\,,$$

where

$$\begin{aligned}P &=_{df} \lambda l\,.\,[Column]\\ i &=_{df} Nil_{[Column]}\\ j &=_{df} \lambda a\,.\,\lambda l\,.\,\lambda h\,.\,Cons_{[Column]}\,(f_{Attribute}\, a)\, l\\ f_{Attribute} &=_{df} \lambda a\,.\,@_{Column}\,(Id_{Attribute}\, a)\colon Attribute \to Column\,,\end{aligned}$$

and

$$\begin{aligned}@^{-1}_{[Attribute]}\, P\, i\, j\, Nil_{Attribute} &\twoheadrightarrow i\\ @^{-1}_{[Attribute]}\, P\, i\, j\, (Cons_{[Attibute]}\, a\, l) &\twoheadrightarrow j\, a\, l\, (@^{-1}_{[Attribute]}\, P\, i\, j\, l)\,.\end{aligned}$$

3. Let

$$f_{Classes} =_{df} \lambda l\,.\,@^{-1}_{[Class]}\, P\, i\, j\, l\colon [Class] \to [Table]\,,$$

where

$$\begin{aligned}P &=_{df} \lambda l\,.\,[Table]\\ i &=_{df} Nil_{[Table]}\\ j &=_{df} \lambda a\,.\,\lambda l\,.\,\lambda h\,.\,Cons_{[Table]}\,(f_{Class}\, c)\, l\\ f_{Class} &=_{df} \lambda c\,.\,@_{Table}\,(Id_{Class}\, c)\,(f_{Attributes}\,(R_{Attributes}\, c))\colon Class \to Table\,,\end{aligned}$$

and

$$\begin{aligned}@^{-1}_{[Class]}\, P\, i\, j\, Nil_{[Class]} &\twoheadrightarrow i\\ @^{-1}_{[Class]}\, P\, i\, j\, (Cons_{[Class]}\, a\, l) &\twoheadrightarrow j\, a\, l\, (@^{-1}_{[Class]}\, P\, i\, j\, l)\,.\end{aligned}$$

The subscripts on $Cons$ and Nil in this and the previous step are eluded from the following steps because they take up too much space. It should be clear from the context which is which.

4. Let

$$f_{Model} =_{df} \lambda m\,.\,@_{Schema}\,(Id_{Model}\, m)\,(f_{Classes}\,(R_{Classes}\, m))\colon Model \to Schema\,.$$

5. Prove

$$\Delta_3 \vdash Post_{Column}\ a\ (f_{Attribute}\ a)\ .$$

$$\dfrac{\dfrac{\dfrac{Id_{Attribute}\colon Attribute \to Nat \quad \dfrac{\Delta_3 \vdash a\colon Attribute_{(Ass)}}{\Delta_3 \vdash Id_{Attribute}\ a\colon Nat}\ (\to E)}{\Delta_3 \vdash Id_{Attribute}\ a = Id_{Attribute}\ a}\ (II)}{\Delta_3 \vdash Id_{Column}\ (f_{Attribute}\ a) = Id_{Attribute}\ a}\ (=_{abs}) \quad \dfrac{\dfrac{\dfrac{\Delta_3 \vdash Pre_{Attribute}\ a\ \wedge a \in R_{Attributes}\ c_{(Ass)}}{\Delta_3 \vdash Pre_{Attribute}\ a}\ (\wedge E_1)}{\Delta_3 \vdash Id_{Attribute}\ a > 0}\ (=_{df})}{\Delta_3 \vdash Id_{Column}\ (f_{Attribute}\ a) > 0}\ (=_{abs})}{\dfrac{\Delta_3 \vdash Id_{Column}\ (f_{Attribute}\ a) = Id_{Attribute}\ a\ \wedge Id_{Column}\ (f_{Attribute}\ a) > 0}{\Delta_3 \vdash Post_{Column}\ a\ (f_{Attribute}\ a)}\ (=_{df})\ .}\ (\wedge I)$$

6. Prove

$$\Delta_3 \vdash f_{Attribute}\ a \in S_{Columns}\ (f_{Class}\ c)\ . \qquad (5.2)$$

Figure 5.16 shows that the column obtained by applying $f_{Attribute}$ to a, is an element of the list of columns obtained by first applying f_{Class} to c, and then applying $S_{Columns}$ to $f_{Class}\ c$.

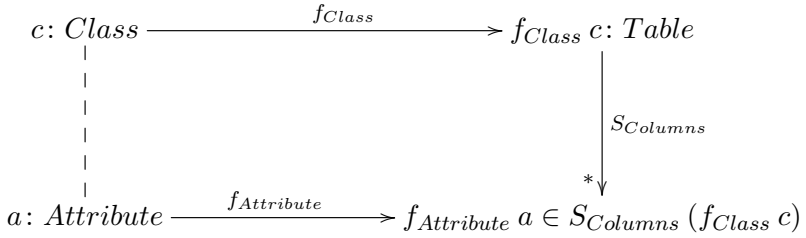

Figure 5.16: A visualisation of the term $f_{Attribute}\ a \in S_{Columns}\ (f_{Class}\ c)$.

Let

$$\begin{aligned}\Delta\ =_{df}\ &\Delta_3,\ [b\colon Attribute,\ l\colon [\,Attribute\,]\,]\ ,\\ &[h_1\colon a \in l \to f_{Attribute}\ a \in f_{Attributes}\ l,\ h_2\colon a \in Cons\ b\ l\,]\ .\end{aligned}$$

There are five parts to the proof.

(a) Prove

$$\Delta_3 \vdash a \in Nil \to f_{Attribute}\ a \in f_{Attributes}\ Nil\ .$$

$$\dfrac{\dfrac{\dfrac{\Delta, [p\colon a \in Nil] \vdash a \in Nil_{(Ass)}}{\Delta, [p\colon a \in Nil] \vdash \bot}\ (=_{df})}{\Delta, [p\colon a \in Nil] \vdash f_{Attribute}\ a \in f_{Attributes}\ Nil}\ (\bot E)}{\Delta_3 \vdash a \in Nil \to f_{Attribute}\ a \in f_{Attributes}\ Nil}\ (\to I)\ .$$

(b) Prove

$$\Delta \vdash a = b \to f_{Attribute}\, a \in f_{Attributes}\, (Cons\, b\, l)\ .$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash f_{Attribute}\colon Attribute \to Column \quad \Delta \vdash a\colon Attribute_{(Ass)}}{\Delta \vdash f_{Attribute}\, a\colon Column}\,(\to E)}{\Delta \vdash f_{Attribute}\, a = f_{Attribute}\, a}\,(II) \qquad \Delta, [p\colon a = b] \vdash a = b_{(Ass)}}{\Delta, [p\colon a = b] \vdash f_{Attribute}\, a = f_{Attribute}\, b}\,(IE)}{\Delta, [p\colon a = b] \vdash f_{Attribute}\, a = f_{Attribute}\, b \vee f_{Attribute}\, a \in f_{Attributes}\, l}\,(\vee I_1)}{\Delta, [p\colon a = b] \vdash f_{Attribute}\, a \in f_{Attributes}\, (Cons\, b\, l)}\,(=_{df})}{\Delta \vdash a = b \to f_{Attribute}\, a \in f_{Attributes}\, (Cons\, b\, l)}\,(\to I)\ .$$

(c) Prove

$$\Delta \vdash a \in b \to f_{Attribute}\, a \in f_{Attributes}\, (Cons\, b\, l)\ .$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta \vdash a \in l \to f_{Attribute}\, a \in f_{Attributes}\, l_{(Ass)} \quad \Delta, [p\colon a \in l] \vdash a \in l_{(Ass)}}{\Delta, [p\colon a \in l] \vdash f_{Attribute}\, a \in f_{Attributes}\, l}\,(\to E)}{\Delta, [p\colon a \in l] \vdash f_{Attribute}\, a = f_{Attribute}\, a \vee f_{Attribute}\, a \in f_{Attributes}\, l}\,(\vee I_2)}{\Delta \vdash a \in b \to f_{Attribute}\, a = f_{Attribute}\, a \vee f_{Attribute}\, a \in f_{Attributes}\, l}\,(\to I)}{\Delta \vdash a \in b \to f_{Attribute}\, a \in Cons\, (f_{Attribute}\, a)\, (f_{Attributes}\, l)}\,(=_{df})}{\Delta \vdash a \in b \to f_{Attribute}\, a \in f_{Attributes}\, (Cons\, b\, l)}\,(=_{abs})\ .$$

(d) Prove

$$\Delta_3 \vdash \forall b\colon Attribute\,.\,\forall l\colon [\,Attribute\,]\,.\,(a \in l \to f_{Attribute}\, a \in f_{Attributes}\, l) \to a \in Cons\, b\, l \to f_{Attribute}\, a \in f_{Attributes}\, (Cons\, b\, l)\ .$$

Use parts (b) and (c).

$$\dfrac{\dfrac{\dfrac{\Delta \vdash a \in Cons\, b\, l_{(Ass)}}{\Delta \vdash a = b \vee a \in l}\,(=_{df}) \quad \Delta \vdash a = b \to f_{Attribute}\, a \in f_{Attributes}\, (Cons\, b\, l) \quad \Delta \vdash a \in b \to f_{Attribute}\, a \in f_{Attributes}\, (Cons\, b\, l)}{\Delta \vdash f_{Attribute}\, a \in f_{Attributes}\, (Con\, b\, l)}\,(\vee E)}{\Delta_3 \vdash \forall b\colon Attribute\,.\,\forall l\colon [\,Attribute\,]\,.\ (a \in l \to f_{Attribute}\, a \in f_{Attributes}\, l) \to a \in Cons\, b\, l \to f_{Attribute}\, a \in f_{Attributes}\, (Con\, b\, l)}\,(\to I)_{\times 2}(\forall I)_{\times 2}\ .$$

(e) Prove (5.2). Use parts (a) and (d).

$$\dfrac{\dfrac{\dfrac{\begin{array}{l}\Delta_3 \vdash a \in Nil \to f_{Attribute}\, a \in f_{Attributes}\, Nil \\ \Delta_3 \vdash \forall b\colon Attribute\,.\,\forall l\colon [\,Attribute\,]\,. \\ \qquad (a \in l \to f_{Attribute}\, a \in f_{Attributes}\, l) \to \\ \qquad\quad a \in Cons\, b\, l \to f_{Attribute}\, a \in f_{Attributes}\, (Con\, b\, l) \\ \Delta_3 \vdash R_{Attributes}\, c\colon [\,Attribute\,]\end{array}}{\Delta_3 \vdash a \in R_{Attributes}\, c \to f_{Attribute}\, a \in f_{Attributes}\, (R_{Attributes}\, c)}\,([\,Attribute\,]\,E) \quad \Delta_3 \vdash a \in R_{Attributes}\, c^{\dagger}}{\dfrac{\Delta_3 \vdash f_{Attribute}\, a \in f_{Attributes}\, (R_{Attributes}\, c)}{\Delta_3 \vdash f_{Attribute}\, a \in S_{Columns}\, (f_{Class}\, c)}\,(=_{abs})}\,(\to E)\ ,$$

where

$$\dfrac{\Delta_3 \vdash Pre_{Attribute}\, a\ \wedge a \in R_{Attributes}\, c_{(Ass)}}{\Delta_3 \vdash a \in R_{Attributes}\, c^{\dagger}}\,(\wedge E_2)\ .$$

7. Prove

$$\Delta_2 \vdash (\forall a\colon Attribute\,.\,Pre_{Attribute}\, a\ \wedge a \in R_{Attributes}\, c \to \ldots\,[(f_{Class}\, c)/t]\ .$$

Use steps 5 and 6.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\Delta_3 \vdash Post_{Column}\, a\, (f_{Attribute}\, a) \\ \Delta_3 \vdash (f_{Attribute}\, a) \in S_{Columns}\, (f_{Class}\, c)\end{array}}{\Delta_3 \vdash Post_{Column}\, a\, (f_{Attribute}\, a)\ \wedge (f_{Attribute}\, a) \in S_{Columns}\, (f_{Class}\, c)}\,(\wedge I)}{\Delta_3 \vdash \exists o\colon Column\,.\,Post_{Column}\, a\, o\ \wedge o \in S_{Columns}\, (f_{Class}\, c)}\,(\exists I)}{\Delta_2 \vdash (\forall a\colon Attribute\,.\,Pre_{Attribute}\, a\ \wedge a \in R_{Attributes}\, c \to \ldots\,[(f_{Class}\, c)/t]}\,(\forall I)(\to I)\ .$$

8. Prove

$$\Delta_2 \vdash Post_{Table}\, c\, (f_{Class}\, c)\ .$$

The proof is similar to that of step 5.

9. Prove

$$\Delta_2 \vdash f_{Class}\, c \in S_{Tables}\, (f_{Model}\, m)\ .$$

The proof is similar to that of step 6.

10. Prove

$$\Delta_1 \vdash (\forall c\colon Class\,.\,Pre_{Class}\, c\ \wedge c \in R_{Classes}\, m \to \ldots)\,[(f_{Model}\, m)/s]\ .$$

Use steps 7, 8 and 9.

$$\dfrac{\dfrac{\dfrac{\begin{array}{l}\Delta_2 \vdash Post_{Table}\, c\, (f_{Class}\, c) \\ \Delta_2 \vdash (f_{Class}\, c) \in S_{Tables}\, (f_{Model}\, m) \\ \Delta_2 \vdash (\forall a\colon Attribute\,.\, Pre_{Attribute}\, a\ \wedge a \in R_{Attributes}\, c \rightarrow \ldots [(f_{Class}\, c)/t]\end{array}}{\begin{array}{l}\Delta_2 \vdash Post_{Table}\, c\, (f_{Class}\, c)\ \wedge (f_{Class}\, c) \in S_{Tables}\, (f_{Model}\, m) \wedge \\ \qquad (\forall a\colon Attribute\,.\, Pre_{Attribute}\, a\ \wedge a \in R_{Attributes}\, c \rightarrow \ldots [(f_{Class}\, c)/t]\end{array}}\,(\wedge I)_{\times 2}}{\Delta_2 \vdash (\exists t\colon Table\,.\, Post_{Table}\, c\, t\ \wedge \ldots)\, [(f_{Model}\, m)/s]}\,(\exists I)}{\Delta_1 \vdash (\forall c\colon Class\,.\, Pre_{Class}\, c\ \wedge c \in R_{Classes}\, m \rightarrow \ldots)\, [(f_{Model}\, m)/s]}\,(\forall I)(\rightarrow I)\ .$$

11. Prove

$$\Delta_1 \vdash Post_{Schema}\, m\, (f_{Model}\, m)\ .$$

The proof is similar to that of step 5.

12. Finally, prove (5.1) using steps 10 and 11.

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{l}\Delta_1 \vdash Post_{Schema}\, m\, (f_{Model}\, m) \\ \Delta_1 \vdash (\forall c\colon Class\,.\, Pre_{Class}\, c\ \wedge c \in R_{Classes}\, m \rightarrow \ldots)\, [(f_{Model}\, m)/s]\end{array}}{\Delta_1 \vdash (Post_{Schema}\, m\, s\ \wedge \forall c\colon Class\,.\, Pre_{Class}\, c\ \wedge c \in R_{Classes}\, m \rightarrow \ldots)\, [(f_{Model}\, m)/s]}\,(\wedge I)}{\Delta_1 \vdash \exists s\colon Schema\,.\, Post_{Schema}\, m\, s\ \wedge \ldots}\,(\exists I)}{\vdash \forall m\colon Model\,.\, Pre_{Model}\, m \rightarrow \ldots}\,(\forall I)(\rightarrow I)}{\vdash Log_{Tran}\, Model\, Schema\, t_{ModelSchema}}\,(=_{abs})\ .$$

□

If $UmlSql$ is the object of $Log_{Tran}\, Model\, Schema\, t_{ModelSchema}$ in step 12 above, then $UmlSql$ is the certified program that implements the transformation. However, given its size, it is best to leave it to Coq to derive.

Coq Listing 58. *(Certified Program)* The certified program below is a function `UmlSql` that takes an object `m` of `Model`, and an unspecified proof (denoted by the underscore symbol in line 2) that `m` satisfies `Pre_Model`, and returns in line 14 the corresponding object of `Schema`, i.e. `(f_Model m)`, embedded within a proof that `m` and `(f_Model m)` satisfy the logical interpretation of $t_{ModelSchema}$, as given in the last line.

```
UmlSql =
fun (m : Model) (_ : Pre_Model m) =>
ex_intro
  (fun y : Schema =>
   Post_Schema m y /\
   (forall x' : Class,
    Pre_Class x' /\ In x' (R_Classes m) ->
    exists y' : Table,
      Post_Table x' y' /\
      In y' (S_Tables y) /\
```

```
    (forall x'0 : Attribute,
     Pre_Attribute x'0 /\ In x'0 (R_Attributes x') ->
     exists y'0 : Column, Post_Column x'0 y'0 /\ In y'0 (S_Columns y'))))
(f_Model m)
(conj (eq_refl (Id_Model m))
   (fun (c : Class) (H2 : Pre_Class c /\ In c (R_Classes m)) =>
    ex_intro
      (fun y' : Table =>
       Post_Table c y' /\
       In y' (S_Tables (f_Model m)) /\
       (forall x' : Attribute,
        Pre_Attribute x' /\ In x' (R_Attributes c) ->
        exists y'0 : Column, Post_Column x' y'0 /\ In y'0 (S_Columns y')))
      (f_Class c)
      match H2 with
      | conj H H0 =>
          conj (conj (eq_refl (Id_Class c)) H)
            (conj
               (let l := R_Classes m in
                let H3 :=
                  list_ind
                    (fun l0 : list Class =>
                     In c l0 -> In (f_Class c) (f_Tables l0))
                    (fun H3 : In c nil =>
                     False_ind (In (f_Class c) (f_Tables nil)) H3)
                    (fun (a : Class) (l0 : list Class)
                       (IHl : In c l0 -> In (f_Class c) (f_Tables l0))
                       (H3 : In c (a :: l0)) =>
                     match H3 with
                     | or_introl H4 =>
                         or_introl (In (f_Class c) (f_Tables l0))
                           (eq_ind_r
                              (fun a0 : Class => f_Class a0 = f_Class c)
                              (eq_refl (f_Class c)) H4)
                     | or_intror H4 =>
                         or_intror (f_Class a = f_Class c) (IHl H4)
                     end) l in
                H3 H0)
               (fun (a : Attribute)
                  (H3 : Pre_Attribute a /\ In a (R_Attributes c)) =>
                ex_intro
                  (fun y' : Column =>
                   Post_Column a y' /\ In y' (S_Columns (f_Class c)))
                  (f_Attribute a)
                  match H3 with
                  | conj H4 H5 =>
                      conj (conj (eq_refl (Id_Attribute a)) H4)
                        (let l := R_Attributes c in
                         let H6 :=
```

```
                        list_ind
                          (fun l0 : list Attribute =>
                           In a l0 ->
                           In (f_Attribute a) (f_Attributes l0))
                          (fun H6 : In a nil =>
                           False_ind
                             (In (f_Attribute a) (f_Attributes nil)) H6)
                          (fun (a0 : Attribute) (l0 : list Attribute)
                             (IH1 : In a l0 ->
                                    In (f_Attribute a) (f_Attributes l0))
                             (H6 : In a (a0 :: l0)) =>
                           match H6 with
                           | or_introl H7 =>
                               eq_ind_r
                                 (fun a1 : Attribute =>
                                  In (f_Attribute a)
                                    (f_Attributes (a1 :: l0)))
                                 (or_introl
                                    (In (f_Attribute a)
                                       (f_Attributes l0))
                                    (eq_refl (f_Attribute a))) H7
                           | or_intror H7 =>
                               or_intror
                                 (f_Attribute a0 = f_Attribute a)
                                 (IH1 H7)
                           end) l in
                      H6 H5)
             end))
   end))
: LogTran Model Schema t_Model_Schema
```

6

Certified Ordered Model Transformations

In virtue of the regular structure of ordered model transformations, it seems reasonable to propose that the proof of the logical interpretation of an ordered model transformation is susceptible to the principle of divide and conquer, whereby the proof of the whole is composed of the sum of the proofs of its parts. The purpose of this chapter is to establish the truth of this proposition. It starts by introducing the Skolemised forms of the logical interpretations of data and link components, and continues by showing how they can be encapsulated within an assembly of components which are the certified analogues of the uncertified components defined in the previous chapter. It concludes with a type theoretical specification and proof of the proposition, followed by a detailed worked example.

6.1 Skolemised Forms

An ordered model transformation is essentially a composition of data and link components. Take, for instance, the three ordered model transformations defined in Section 5.2.7.

- t^1_{AW} is the composition of a solitary data component d_{AW}, i.e.

$$@^1_{Tran}\ A\ W\ d_{AW}\ ;$$

- t^2_{AW} is the composition of four data components d_{AW}, d_{BX}, d_{CY} and d_{DZ}, and three link components l_{AW}, l^1_{BX} and l^2_{BX}, i.e.

$$\begin{aligned}@^2_{Tran}\ A\ W\ d_{AW}\ (@^2_{Poset}\ A\ W\ B\ X\ (@_{DataLink}\ A\ W\ B\ X\ d_{BX}\ l_{AW})\\ (@^3_{Poset}\ B\ X\ (@^1_{Poset}\ B\ X\ C\ Y\ (@_{DataLink}\ B\ X\ C\ Y\ d_{CY}\ l^1_{BX}))\\ (@^1_{Poset}\ B\ X\ D\ Z\ (@_{DataLink}\ B\ X\ D\ Z\ d_{DZ}\ l^2_{BX}))))\ ;\end{aligned}$$

- t^3_{AW} is the composition of two data components d_{AW} and d_{BX}, and one link component l_{AW}, i.e.

$$@^2_{Tran}\ A\ W\ d_{AW}\ (@^1_{Poset}\ A\ W\ B\ X\ (@_{DataLink}\ A\ W\ B\ X\ d_{BX}\ l_{AW}))\ .$$

On the basis of the structure of, say, t^3_{AW}, it seems reasonable to propose:

Proposition 1. *The logical interpretation of (the whole of)* t^3_{AW} *is provable if the logical interpretations of (the parts of* t^3_{AW}*)* d_{AW}*,* d_{BX} *and* l_{AW} *are provable, i.e.*

$$\begin{aligned}\vdash\ & Log_{Data}\ A\ W\ d_{AW} \rightarrow \\ & Log_{Data}\ B\ X\ d_{BX} \rightarrow \\ & \forall a\colon A\,.\,\forall w\colon W\,.\,Log_{Link}\ A\ W\ B\ X\ l_{AW}\ a\ w \rightarrow \\ & \qquad Log_{Tran}\ A\ W\ t^3_{AW}\ .\end{aligned} \tag{6.1}$$

However, this proposition is not provable. When eliminated, the existential quantifiers in the hypotheses of the proof of (6.1) (see the definitions of Log_{Data} and Log_{Link} in Section 5.3.2) produce anonymous witnesses which are simply not useful in proving the conclusion. A suitable way round this problem is to Skolemise [130] the logical interpretations of the data and link components, so as to remove the existential quantifiers.

Definition 57. *(Skol)* The Skolemised form of the logical interpretation of a data component d on X and Y—that is to say the formula obtained by removing the existential quantifier from the logical interpretation of d, and replacing the existential variable with the term $(f\,x)$, where f is a Skolem function that takes X to Y, and x is an object of X—is given by

$$Skol\ X\ Y\ d\ f\ ,$$

where

$$\begin{aligned} Skol &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda d\,.\,@^{-1}_{Data}\ P\ X\ Y\ d\ i \\ P &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda d\,.\,(X \rightarrow Y) \rightarrow U_0 \\ i &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda Pre\,.\,\lambda Post\,.\,\lambda f\,.\,\forall x\colon X\,.\,Pre\ x \rightarrow Post\ x\ (f\ x)\ . \end{aligned}$$

Example 22. *(Skol)* The Skolemised form of the logical interpretation of d_{AW} is

$$\begin{aligned} Skol\ A\ W\ d_{AW}\ f_A &\twoheadrightarrow i\ A\ W\ Pre_A\ Post_W\ f_A \\ &\twoheadrightarrow \forall x\colon A\,.\,Pre_A\ x \rightarrow Post_W\ x\ (f_A\ x) \\ &\rightarrow_\alpha \forall a\colon A\,.\,Pre_A\ a \rightarrow Post_W\ a\ (f_A\ a)\ , \end{aligned}$$

where f_A is the Skolem function.

Coq Listing 59. *(Skol)*

```
Definition Skol (X : Set) (Y : Set) (d : Data X Y) : (X -> Y) -> Prop :=
   match d with Build_Data X Y Pre Post =>
      fun f : X -> Y => forall x : X, Pre x -> Post x (f x)
   end.
```

The relationship between the logical interpretation of a data component and its Skolemised form, is defined by the following theorem.

Theorem 2. *(Skolemised Data Component)* The logical interpretation of a data component is provable if its Skolemised form is provable, i.e.

$$\vdash \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall d\colon Data\ X\ Y\,.\,\forall f\colon X \rightarrow Y\,.\,Skol\ X\ Y\ d\ f \rightarrow Log_{Data}\ X\ Y\ d\ .$$

Proof. The proof requires a reduction at one end, and a data elimination and an expansion at the other end. To save space, the types of most variables are elided.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[X,\,Y,\,Pre,\,Post,\,f,\,s\colon Skol\;X\;Y\;(@_{Data}\;X\;Y\;Pre\;Post)\;f\,]\vdash Skol\;X\;Y\;(@_{Data}\;X\;Y\;Pre\;Post)\;f_{(Ass)}}\;(=_{red})}{[X,\,Y,\,Pre,\,Post,\,f,\,s\,]\vdash\forall x\colon X\,.\,Pre\;x\to Post\;x\;(f\;x)}\;(\forall E)(\to E)}{[X,\,Y,\,Pre,\,Post,\,f,\,s,\,x,\,h\,]\vdash Post\;x\;(f\;x)}\;(\exists\,I)}{[X,\,Y,\,Pre,\,Post,\,f,\,s,\,x,\,h\,]\vdash\exists y\colon Y\,.\,Post\;x\;y}\;(\to I)\,(\forall I)}{[X,\,Y,\,Pre,\,Post,\,f,\,s\,]\vdash\forall x\colon X\,.\,Pre\;x\to\exists y\colon Y\,.\,Post\;x\;y}\;(=_{exp})}{[X,\,Y,\,Pre,\,Post,\,f,\,s\,]\vdash Log_{Data}\;X\;Y\;(@_{Data}\;X\;Y\;Pre\;Post)}\;(\to I)\,(\forall I)_*}{\vdash\forall X\,.\,\forall Y\,.\,\forall Pre\,.\,\forall Post\,.\,\forall f\,.\,Skol\;X\;Y\;(@_{Data}\;X\;Y\;Pre\;Post)\;f\to Log_{Data}\;X\;Y\;(@_{Data}\;X\;Y\;Pre\;Post)}\;(Data\,E_1)}{[X,\,Y,\,d\,]\vdash\forall f\,.\,Skol\;X\;Y\;d\;f\to Log_{Data}\;X\;Y\;d}\;(\forall I)_*\,.$$

□

Applying Theorem 2 to d_{AW}, and taking the Skolem function to be f_A, gives

$$Skol\;A\;W\;d_{AW}\;f_A\to Log_{Data}\;A\;W\;d_{AW}\;,$$

i.e. the logical interpretation of d_{AW} is provable if the Skolemised form of the logical interpretation of d_{AW} is provable. Similarly, it can be shown that

$$Skol\;B\;X\;d_{BX}\;f_B\to Log_{Data}\;B\;X\;d_{BX}$$

for some f_B.

Definition 58. *(Com)* The Skolemised form of the logical interpretation of a link component l on X, Y, X' and Y', on a Skolem function f that takes X to Y, and an auxilliary function f' that takes X' to Y', is given by

$$Com\;X\;Y\;X'\;Y'\;l\;f\;f'\;,$$

where

$$\begin{aligned}
Com &=_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda l\,.\,\lambda f\,.\,\lambda f'\,.\,@^{-1}_{Link}\;P\;X\;Y\;X'\;Y'\;l\;i\;j\\
P &=_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda l\,.\,(X\to Y)\to(X'\to Y')\to U_0\\
i &=_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda r\,.\,\lambda s\,.\,\lambda f\,.\,\lambda f'\,.\,\forall x\colon X\,.\,f'\;(r\;x)=s\;(f\;x)\\
j &=_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda r\,.\,\lambda s\,.\,\lambda f\,.\,\lambda f'\,.\,\forall x\colon X\,.\,\forall x'\colon X'\,.\,x'\in r\;x\to f'\;x'\in s\;(f\;x)\;.
\end{aligned}$$

Example 23. *(Com)* The Skolemised form of the logical interpretation of l_{AW} is

$$\begin{aligned} Com\ A\ W\ B\ X\ l_{AW}\ f_A\ f_B &\twoheadrightarrow i\ A\ W\ B\ X\ R_1\ S_1\ f_A\ f_B \\ &\twoheadrightarrow \forall x\colon A\,.\,f_B\ (R_1\ x) = S_1\ (f_A\ x) \\ &\rightarrow_\alpha \forall a\colon A\,.\,f_B\ (R_1\ a) = S_1\ (f_A\ a)\ , \end{aligned}$$

where f_A is the Skolem function, and f_B is the auxiliary function. Since l_{AW} binds together d_{AW} and d_{BX}, the Skolem function of l_{AW} must be the same as the Skolem function of d_{AW}, and the auxilliary function of l_{AW} must be the same as the Skolem function of d_{BX}.

Coq Listing 60. *(Com)*

```
Definition Com (X : Set) (Y : Set) (X' : Set) (Y' : Set) (l : Link X Y X' Y')
      : (X -> Y) -> (X' -> Y') -> Prop :=
   match l with
      Build_Link_1 X Y X' Y' r s =>
         fun (f : X -> Y) (f' : X' -> Y') =>
            forall x : X, f' (r x) = s (f x) |
      Build_Link_2 X Y X' Y' r s =>
         fun (f : X -> Y) (f' : X' -> Y') =>
            forall x : X, forall x' : X', In x' (r x) -> In (f' x') (s (f x))
      end.
```

The relationship between the logical interpretation of a link component and its Skolemised form, is defined by the following theorem.

Theorem 3. *(Skolemised Link Component)* The logical interpretation of a link component l on X, Y, X' and Y', is provable on all objects x of X and $(f\ x)$ of Y, if the Skolemised form of the logical interpretation of l on f and any auxilliary function f', is provable, i.e.

$$\begin{aligned} &\vdash \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,\forall l\colon Link\ X\ Y\ X'\ Y'\,.\,\forall f\colon X \to Y\,.\,\forall f'\colon X' \to Y'\,. \\ &\quad Com\ X\ Y\ X'\ Y'\ l\ f\ f' \to \forall x\colon X\,.\,Log_{Link}\ X\ Y\ X'\ Y'\ l\ x\ (f\ x)\ . \end{aligned} \tag{6.2}$$

Proof. The proof is in three steps.

1. Prove

$$\begin{aligned} &\vdash \forall X\,.\,\forall Y\,.\,\forall X'\,.\,\forall Y'\,.\,\forall R\,.\,\forall S\,.\,\forall f\,.\,\forall f'\,. \\ &\quad Com\ X\ Y\ X'\ Y'\ (@^1_{Link}\ \ldots)\ f\ f' \to \forall x\,.\,Log_{Link}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ \ldots)\ x\ (f\ x)\ . \end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\,X,\,Y,\,X',\,Y',\,R,\,S,\,f,\,f',\,c\colon Com\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ R\ S)\ f\ f'\,] \vdash Com\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ R\ S)\ f\ f'}\ (Ass)}{[\,X,\,Y,\,X',\,Y',\,R,\,S,\,f,\,f',\,c\,] \vdash \forall x\colon X\,.\,f'\ (R\ x) = S\ (f\ x)}\ (=_{red}) \\ \dfrac{[\,X,\,x\,] \vdash x_{(Ass)}}{[\,X,\,Y,\,X',\,Y',\,R,\,S,\,f,\,f',\,c,\,x\,] \vdash f'\ (R\ x) = S\ (f\ x)}\ (\forall E) \\ \dfrac{}{[\,X,\,Y,\,X',\,Y',\,R,\,S,\,f,\,f',\,c,\,x\,] \vdash \exists y'\colon Y'\,.\,y' = S\ (f\ x)}\ (\exists\, I) \\ [\,X,\,Y,\,X',\,Y',\,R,\,S,\,f,\,f',\,c,\,x\,] \vdash Log_{Link}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ \ldots)\ x\ (f\ x)\ (=_{abs}) \\ \vdash \forall X\,.\,\forall Y\,.\,\forall X'\,.\,\forall Y'\,.\,\forall R\,.\,\forall S\,.\,\forall f\,.\,\forall f'\,. \\ \quad Com\ X\ Y\ X'\ Y'\ (@^1_{Link}\ \ldots)\ f\ f' \to \forall x\,.\,Log_{Link}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ \ldots)\ x\ (f\ x) \end{aligned}\ (\forall I)_*\ .$$

2. Prove

$$\vdash \forall X\,.\,\forall Y\,.\,\forall X'\,.\,\forall Y'\,.\,\forall R\,.\,\forall S\,.\,\forall f\,.\,\forall f'\,.$$
$$Com\ X\ Y\ X'\ Y'\ (@^1_{Link}\ldots)\ f\ f' \to \forall x\,.\,Log_{Link}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ldots)\ x\ (f\ x)\ .$$

The proof is similar to step 1, except that it relates to $@^2_{Link}$ instead of $@^1_{Link}$.

3. Prove (6.2).

$$\frac{\begin{array}{l}\vdash \forall X\,.\,\forall Y\,.\,\forall X'\,.\,\forall Y'\,.\,\forall R\,.\,\forall S\,.\\ \forall f\,.\,\forall f'\,.\,Com\ X\ Y\ X'\ Y'\ (@^1_{Link}\ldots)\ f\ f' \to \forall x\,.\,Log_{Link}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ldots)\ x\ (f\ x)\\ \vdash \forall X\,.\,\forall Y\,.\,\forall X'\,.\,\forall Y'\,.\,\forall R\,.\,\forall S\,.\\ \forall f\,.\,\forall f'\,.\,Com\ X\ Y\ X'\ Y'\ (@^2_{Link}\ldots)\ f\ f' \to \forall x\,.\,Log_{Link}\ X\ Y\ X'\ Y'\ (@^2_{Link}\ldots)\ x\ (f\ x)\end{array}}{\begin{array}{l}[X,\,Y,\,X',\,Y',\,l\,]\vdash\\ \forall f\,.\,\forall f'\,.\,Com\ X\ Y\ X'\ Y'\ l\ f\ f' \to \forall x\,.\,Log_{Link}\ X\ Y\ X'\ Y'\ l\ x\ (f\ x)\end{array}}\ (Link\,E_1)$$
$$\frac{}{\begin{array}{l}\vdash \forall X\,.\,\forall Y\,.\,\forall X'\,.\,\forall Y'\,.\,\forall l\,.\,\forall f\,.\,\forall f'\,.\\ Com\ X\ Y\ X'\ Y'\ l\ f\ f' \to \forall x\,.\,Log_{Link}\ X\ Y\ X'\ Y'\ l\ x\ (f\ x)\end{array}}\ (\forall I)_*\ .$$

□

Applying Theorem 3 to l_{AW} gives

$$Com\ A\ W\ B\ X\ l_{AW}\ f_A\ f_B \to \forall x\,.\,Log_{Link}\ A\ W\ B\ X\ l_{AW}\ x\ (f_A\ x)\ ,$$

i.e. the logical interpretation of l_{AW} is provable on all objects x of A and $(f_A\ x)$ of W, if the Skolomised form of l_{AW} on f_A and some f_B is provable.

It was remarked earlier that although Proposition 1 is unprovable, it can easily be turned into one that is provable by replacing the three hypotheses with their Skolomised forms. This is indeed the case.

Proposition 2. *The logical interpretation of (the whole of) t^3_{AW} is provable if the Skolomised forms of the logical interpretations of (the parts of t^3_{AW}) d_{AW}, d_{BX} and l_{AW} are provable, i.e.*

$$\vdash Skol\ A\ W\ d_{AW}\ f_A \to Skol\ B\ X\ d_{BX}\ f_B \to Com\ A\ W\ B\ X\ l_{AW}\ f_A\ f_B \to$$
$$Log_{Tran}\ A\ W\ t^3_{AW}\ .$$

The proof is omitted. Clearly, the principle of proving the whole by summing the proofs of its parts could be applied on a case by case basis to any *specific* ordered model transformation, no matter how large. However, the challenge (and the aim of this chapter) is to show how this principle can be applied to *all* ordered model transformations in one fell swoop.

6.2 Certified Components

A *certified* component of an ordered model transformation is the analogue of an *uncertified* component of an ordered model transformation, as defined in Section 5.2, although back there it was not called an uncertified component, just a component. As its name suggests, a certified component carries sufficient information to prove the logical interpretation of the certified component.

Definition 59. *(Certified Component)* A certified component $\surd c$, of a component c, is one that carries a proof of the Skolomised form of the logical interpretation of every data component and every link component in c.[1]

There are five kinds of uncertified components, and therefore five kinds of certified components.

6.2.1 Certified Data Component

A certified data component is the composition of an uncertified data component (as defined in Section 5.2.1) and a proof of the Skolemised form of the logical interpretation of the data component (as defined in Section 6.1). Figure 6.1 shows a certified data component on X, Y and f, comprising a data component d, say, on X and Y, a Skolem function f, and a proof $\mathfrak{S}$ of the Skolemised form of the logical interpretation of d.

$$X \overset{Pre}{\vdash\!\!-\!\!-} \frac{f}{\mathfrak{S}} \overset{Post}{-\!\!-\!\!\succ} Y$$

Figure 6.1: A certified data component on X, Y and f.

Definition 60. *(Certified Data Component)* A certified data component on a source class X, a target class Y, and a function f that takes X to Y, is an inhabitant of type $\surd Data\ X\ Y\ f$.

Formation Rule

The $\surd Data$ formation rule, which is given by

$$\frac{X : U_0 \quad Y : U_0 \quad f : X \to Y}{\surd Data\ X\ Y\ f : U_1}\ (\surd Data\ F)\ ,$$

asserts that $\surd Data\ X\ Y\ f$ is an inhabitant of U_1, if X and Y are inhabitants of U_0, and f is a function that takes X to Y.

[1] The symbol $\surd$ denotes "certified" thoughout.

Introduction Rule

The $\surd Data$ introduction rule, which is given by

$$\frac{\begin{array}{lll} X\colon U_0 & Y\colon U_0 & f\colon X \to Y \end{array} \quad \begin{array}{l} d\colon Data\ X\ Y \\ s\colon Skol\ X\ Y\ d\ f \end{array}}{@_{\surd Data}\ X\ Y\ f\ d\ s\colon \surd Data\ X\ Y\ f}\ (\surd Data\ I)\ ,$$

asserts that $(@_{\surd Data}\ X\ Y\ f\ d\ s)$ is a certified data component on X, Y and f, if d is a data component on X and Y, and s is a proof of the Skolemised form of the logical interpretation of d on f. For example, if f_A is a function that takes A to W, d_{AW} is a data component on A and W (see Section 5.2.7), and s_{AW} is a proof of the Skolemised form of the logical interpretation of d_{AW}, then the certified data component $\surd d_{AW}$ on A, W and f_A is given by

$$\frac{\begin{array}{lll} A\colon U_0 & W\colon U_0 & f_A\colon A \to W \end{array} \quad \begin{array}{l} d_{AW}\colon Data\ A\ W \\ s_{AW}\colon Skol\ A\ W\ d_{AW}\ f_A \end{array}}{(\surd d_{AW}\ =_{df}\ @_{\surd Data}\ A\ W\ f_A\ d_{AW}\ s_{AW})\colon \surd Data\ A\ W\ f_A}\ (\surd Data\ I)\ .$$

Similarly,

$$(\surd d_{BX}\ =_{df}\ @_{\surd Data}\ B\ X\ f_B\ d_{BX}\ s_{BX})\colon \surd Data\ A\ W\ f_A\ .$$

Elimination Rule

The $\surd Data$ elimination rules are given by

$$\frac{\begin{array}{l} P\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall f\colon X \to Y\,.\,(\surd Data\ X\ Y\ f \to U_n) \\ X\colon U_0 \\ Y\colon U_0 \\ f\colon X \to Y \\ \surd d\colon \surd Data\ X\ Y\ f \\ i\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall f\colon X \to Y\,.\,\forall d\colon Data\ X\ Y\,.\,\forall s\colon Skol\ X\ Y\ d\ f\,. \\ \qquad P\ X\ Y\ f\ (@_{\surd Data}\ X\ Y\ f\ d\ s) \end{array}}{@^{-1}_{\surd Data}\ P\ X\ Y\ f\ \surd d\ i\colon P\ X\ Y\ f\ \surd d}\ (\surd Data\ E_n)\ ,$$

for $n = 0, 1, 2, \ldots$. Each rule asserts, for a particular universe U_n, that $P\ X\ Y\ f\ \surd d$ is inhabited for all certified data components $\surd d$, if $P\ X\ Y\ f\ (@_{\surd Data}\ X\ Y\ f\ d\ s)$ is inhabited for all data components d, and all proofs s of the Skolemised form of the logical interpretation of d on f. The elimination rules are useful in proving properties of certified data components.

Computation Rule

The $\surd Data$ computation rule, which is given by

$$@^{-1}_{\surd Data}\ P\ X\ Y\ f\ (@_{\surd Data}\ X\ Y\ f\ d\ s)\ i \twoheadrightarrow i\ X\ Y\ f\ d\ s\ ,$$

asserts that the term on the right is a simplification of the term on the left, in the sense that it is structurally smaller. Both terms have the same type, i.e. $P\ X\ Y\ f\ (@_{\surd Data}\ X\ Y\ f\ d\ s)$.

Coq Listing 61. *($\surd Data$)*

```
Inductive CertData : forall X : Set, forall Y : Set, (X -> Y) -> Type :=
   Build_CertData :
      forall X : Set,
      forall Y : Set,
      forall f : X -> Y,
      forall d : Data X Y,
      (Skol X Y d) f ->
         CertData X Y f.
```

6.2.2 Certified Link Component

A certified link component is the composition of an uncertified link component (as defined in Section 5.2.2) and a proof of the Skolomised form of the logical interpretation of the link component (as defined in Section 6.1). Figure 6.2 shows a certified link component on X, Y, f, X', Y' and f' (where f and f' are Skolem functions that take X to Y, and X' and Y' respectively), comprising a link component l, say, on X, Y, X' and Y', and a proof $\mathfrak{S}$ of the Skolomised form of the logical interpretation of l on f and f'.

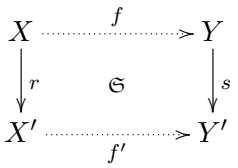

Figure 6.2: A certified link component on X, Y and f.

Definition 61. *(Certified Link Component)* A certified link component on a source class X, a target class Y, a function f that takes X to Y, and primed versions of the same, is an inhabitant of type $\surd Link\ X\ Y\ f\ X'\ Y'\ f'$.

Formation Rule

The $\surd Link$ formation rule, which is given by

$$\frac{\begin{array}{ll} X : U_0 & X' : U_0 \\ Y : U_0 & Y' : U_0 \\ f : X \to Y & f' : X' \to Y' \end{array}}{\surd Link\ X\ Y\ f\ X'\ Y'\ f' : U_1}\ (\surd Link\ F)\ ,$$

asserts that $\surd Link\ X\ Y\ f\ X'\ Y'\ f'$ is an inhabitant of U_1, if X, Y, X' and Y' are inhabitants of U_0, f is a function that takes X to Y, and f' is a function that takes X' to Y'.

Introduction Rule

The $\surd Link$ introduction rule, which is given by

$$\frac{\begin{array}{lll} X\colon U_0 & X'\colon U_0 & \\ Y\colon U_0 & Y'\colon U_0 & l\colon Link\ X\ Y\ X'\ Y' \\ f\colon X \to Y & f'\colon X' \to Y' & c\colon Com\ X\ Y\ X'\ Y'\ l\ f\ f' \end{array}}{@_{\surd Link}\ X\ Y\ f\ X'\ Y'\ f'\ l\ c\colon \surd Link\ X\ Y\ f\ X'\ Y'\ f'}\ (\surd Link\ I)\ ,$$

asserts that $(@_{\surd Link}\ X\ Y\ f\ X'\ Y'\ f'\ l\ c)$ is a certified link component on X, Y, f, X', Y' and f', if l is a link component on X, Y, X' and Y', and c is a proof of the Skolomised form of the logical interpretation of l on f and f'. For example, if f_A is a function that takes A to W, f_B is a function that takes B to X, l_{AW} is a link component on A, W, B and X (as defined in Section 5.2.7), and c_{AW} is a proof of the Skolomised form of the logical interpretation of l_{AW} on f_A and f_B, then the certified link component $\surd l_{AW}$ on A, W, B and X is given by

$$\frac{\begin{array}{lll} A\colon U_0 & B\colon U_0 & \\ W\colon U_0 & X\colon U_0 & l_{AW}\colon Link\ A\ W\ B\ X \\ f_A\colon A \to W & f_B\colon B \to X & c_{AW}\colon Com\ A\ W\ B\ X\ l_{AW}\ f_A\ f_B \end{array}}{(\surd l_{AW}\ =_{df}\ @_{\surd Link}\ A\ W\ f_A\ B\ X\ f_B\ l_{AW}\ c_{AW})\colon \surd Link\ X\ Y\ f\ X'\ Y'\ f'}\ (\surd Link\ I)\ .$$

Elimination Rules

The $\surd Link$ elimination rules are given by

$$\frac{\begin{array}{l} P\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall f\colon X \to Y\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,\forall f'\colon X' \to Y'\,. \\ \quad \surd Link\ X\ Y\ f\ X'\ Y'\ f' \to U_n \\ X\colon U_0 \\ Y\colon U_0 \\ f\colon X \to Y \\ X'\colon U_0 \\ Y'\colon U_0 \\ f'\colon X' \to Y' \\ \surd l : \surd Link\ X\ Y\ f\ X'\ Y'\ f' \\ i\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall f\colon X \to Y\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,\forall f'\colon X' \to Y'\,. \\ \quad \forall l\colon Link\ X\ Y\ X'\ Y'\,.\,\forall c\colon Com\ X\ Y\ X'\ Y'\ l\ f\ f'\,. \\ \quad\quad P\ X\ Y\ f\ X'\ Y'\ f'\ (@_{\surd Link}\ X\ Y\ f\ X'\ Y'\ f'\ l\ c) \end{array}}{@^{-1}_{\surd Link}\ P\ X\ Y\ f\ X'\ Y'\ f'\ \surd l\ i\colon P\ X\ Y\ f\ X'\ Y'\ f'\ \surd l}\ (\surd Link\ E_n)\ ,$$

for $n = 0, 1, 2, \ldots$. Each rule asserts, for a particular universe U_n, that $P\ X\ Y\ f\ X'\ Y'\ f'\ \surd l$ is inhabited for all certified link components $\surd l$, if

$$P\ X\ Y\ f\ X'\ Y'\ f'\ (@_{\surd Link}\ X\ Y\ f\ X'\ Y'\ f'\ l\ c)$$

is inhabited for all link components l, and all proofs c of the Skolomised form of the logical interpretation of l on f and f'.

Computation Rule

The $\surd Link$ computation rule is given by

$$@^{-1}_{\surd Link}\ P\ X\ Y\ f\ X'\ Y'\ f'\ (@_{\surd Link}\ X\ Y\ f\ X'\ Y'\ f'\ l\ c)\ i \twoheadrightarrow i\ X\ Y\ f\ X'\ Y'\ f'\ l\ c\ .$$

Coq Listing 62. *($\surd Link$)*

```
Inductive CertLink : forall X : Set, forall Y : Set, (X -> Y) ->
   forall X' : Set, forall Y' : Set, (X' -> Y') -> Type :=

   Build_CertLink :
      forall X : Set,
      forall Y : Set,
      forall f : X -> Y,
      forall X' : Set,
      forall Y' : Set,
      forall f' : X' -> Y',
      forall l : Link X Y X' Y',
      (Com X Y X' Y' l) f f' ->
         CertLink X Y f X' Y' f'.
```

6.2.3 Certified Data Link Component

A certified data link component is the composition of a certified data component and a certified link component. Figure 6.3 shows a certified data link component on X, Y, f, X', Y' and f' (where f and f' are Skolem functions that take X to Y and X' to Y' respectively), comprising a certified data component $\surd d'$ on X', Y' and f', and a certified link component $\surd l$ on X, Y, f, X', Y' and f'.

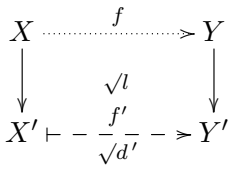

Figure 6.3: A certified data link component on X, Y, f, X', Y' and f'.

Definition 62. *(Certified Data Link Component)* A certified data link component on a source class X, a target class Y, a function f that takes X to Y, and primed versions of the same, is an inhabitant of type $\surd DataLink\ X\ Y\ f\ X'\ Y'\ f'$.

Formation Rule

The $\surd DataLink$ formation rule, which is given by

$$\frac{\begin{array}{ll} X : U_0 & X' : U_0 \\ Y : U_0 & Y' : U_0 \\ f : X \to Y & f' : X' \to Y' \end{array}}{\surd DataLink\ X\ Y\ f\ X'\ Y'\ f' : U_1}\ (\surd DataLink\ F)\ ,$$

asserts that $\surd DataLink\ X\ Y\ f\ X'\ Y'\ f'$ is an inhabitant of U_1, if X, Y, X' and Y' are inhabitants of U_0, f is a function that takes X to Y, and f' is a function that takes X' to Y'.

Introduction Rule

The $\surd DataLink$ introduction rule, which is given by

$$\frac{\begin{array}{llll} X : U_0 & X' : U_0 & \surd d' : \surd Data\ X'\ Y'\ f' \\ Y : U_0 & Y' : U_0 & \surd l : \surd Link\ X\ Y\ f\ X'\ Y'\ f' \\ f : X \to Y & f' : X' \to Y' & \end{array}}{@_{\surd DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd d'\ \surd l : \surd DataLink\ X\ Y\ f\ X'\ Y'\ f'}\ (\surd DataLink\ I)\ ,$$

asserts that $(@_{\surd DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd d'\ \surd l\,)$ is a certified data link component on X, Y, f, X', Y' and f', if $\surd d'$ is a certified data component on X', Y' and f', and $\surd l$ is a certified link component on X, Y, f, X', Y' and f'. For example, if $\surd d_{BX}$ is the certified data component on B, X and f_B, as defined in Section 6.2.1, and $\surd l_{AW}$ is the certified link component on A, W, f_A, B, X and f_B, as defined in Section 6.2.2, then the certified data link component $\surd dl_{AW}$ on A, W, f_A, B, X and f_B is given by

$$\frac{\begin{array}{llll} A : U_0 & B : U_0 & \surd d_{BX} : \surd Data\ B\ X\ f_B \\ W : U_0 & X : U_0 & \surd l_{AW} : \surd Link\ A\ W\ f_A\ B\ X\ f_B \\ f_A : A \to W & f_B : B \to X & \end{array}}{\begin{array}{l} (\surd dl_{AW}\ =_{df}\ @_{\surd DataLink}\ A\ W\ f_A\ B\ X\ f_B\ \surd d_{BX}\ \surd l_{AW}\,) : \\ \qquad\qquad \surd DataLink\ A\ W\ f_A\ B\ X\ f_B \end{array}}\ (\surd DataLink\ I)\ .$$

Elimination Rules

The $\surd DataLink$ elimination rules are given by

$$\frac{\begin{array}{l} P\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall f\colon X \to Y\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,\forall f'\colon X' \to Y'\,. \\ \quad \surd DataLink\ X\ Y\ f\ X'\ Y'\ f' \to U_n \\ X\colon U_0 \\ Y\colon U_0 \\ f\colon X \to Y \\ X'\colon U_0 \\ Y'\colon U_0 \\ f'\colon X' \to Y' \\ \surd dl : \surd DataLink\ X\ Y\ f\ X'\ Y'\ f' \\ i\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall f\colon X \to Y\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,\forall f'\colon X' \to Y'\,. \\ \quad \forall \surd d' : \surd Data\ X'\ Y'\ f'\,.\,\forall \surd l : \surd Link\ X\ Y\ f\ X'\ Y'\ f'\,. \\ \quad\quad P\ X\ Y\ f\ X'\ Y'\ f'\ (@_{\surd DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd d'\ \surd l\,) \end{array}}{@^{-1}_{\surd DataLink}\ P\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ i\colon P\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl}\ (\surd DataLink\ E_n)\ ,$$

for $n = 0, 1, 2, \ldots$. Each rule asserts, for a particular universe U_n, that $P X\ Y\ f\ X'\ Y'\ f'\ \surd dl$ is inhabited for all certified data link components $\surd dl$, if

$$P\ X\ Y\ f\ X'\ Y'\ f'\ (@_{\surd DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd d'\ \surd l\,)$$

is inhabited for all certified data components $\surd d'$, and all certified link components $\surd l$.

Computation Rule

The $\surd DataLink$ computation rule, which is given by

$$\begin{array}{l} @^{-1}_{\surd DataLink}\ P\ X\ Y\ f\ X'\ Y'\ f'\ (@_{\surd DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd d'\ \surd l\,)\ i \twoheadrightarrow \\ \qquad\qquad i\ X\ Y\ f\ X'\ Y'\ f'\ \surd d'\ \surd l\ , \end{array}$$

asserts that the term on the right is a simplification of the term on the left. For example, if

$$\begin{array}{rl} P & =_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda \surd dl\ .\,DataLink\ X\ Y\ X'\ Y' \\ i & =_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda \surd d'\ .\,\lambda \surd l\ .\,\surd l\ , \end{array}$$

i.e. i is a function that returns the certified link component of a pair of certified data and link components, then

$$\begin{array}{l} @^{-1}_{\surd DataLink}\ P\ A\ W\ f_A\ B\ X\ f_B\ \surd dl_{AW}\ i \\ =_{df}\ @^{-1}_{\surd DataLink}\ P\ A\ W\ f_A\ B\ X\ f_B\ (@_{\surd DataLink}\ A\ W\ f_A\ B\ X\ f_B\ \surd d_{BX}\ \surd l_{AW}\,)\ i \\ \twoheadrightarrow i\ A\ W\ f_A\ B\ X\ f_B\ \surd d_{BX}\ \surd l_{AW} \\ \twoheadrightarrow \surd l_{AW}\ . \end{array}$$

Coq Listing 63. *($\surd DataLink$)*

```
Inductive CertDataLink : forall X : Set, forall Y : Set, (X -> Y) ->
   forall X' : Set, forall Y' : Set, (X' -> Y') -> Type :=

   Build_CertDataLink :
      forall X : Set,
      forall Y : Set,
      forall f : X -> Y,
      forall X' : Set,
      forall Y' : Set,
      forall f' : X' -> Y',
      CertData X' Y' f' ->
      CertLink X Y f X' Y' f' ->
         CertDataLink X Y f X' Y' f'.
```

6.2.4 Certified Data Link Poset

A certified data link poset is a partially ordered set of certified data link components, where as usual order is defined by containment, in the sense that if $\surd dl_1$ is a certified data link component on X_1, Y_1, f_1, X_1', Y_1' and f_1', and $\surd dl_2$ is a certified data link component on X_2, Y_2, f_2, X_2', Y_2' and f_2', then the proposition "$\surd dl_1$ is less than $\surd dl_2$" holds if and only if X_2 is an ancestor of X_1, and Y_2 is an ancestor of Y_1.

Figure 6.4 shows three certified data link posets on X, Y and f: from left to right, a *base* certified data link poset comprising a single certified data link component $\surd dl$ on X, Y, f, X', Y' and f'; a *step* certified data link poset comprising a certified data link component $\surd dl$ on X, Y, f, X', Y' and f', and a certified data link poset $\surd p'$ on X', Y' and f'; and a *join* certified data link poset comprising two certified data link posets $\surd p_1$ and $\surd p_2$ on X, Y and f.

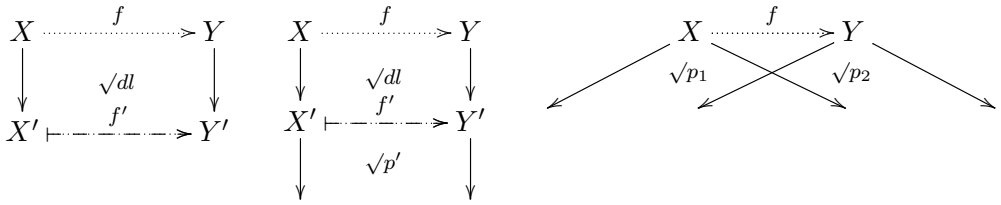

Figure 6.4: Three certified data link posets on X, Y and f.

Definition 63. *(Certified Data Link Poset)* A certified data link poset on a source class X, a target class Y, and a function f that takes X to Y, is an inhabitant of type $\surd Poset\ X\ Y\ f$.

Formation Rule

The $\surd Poset$ formation rule, which is given by

$$\frac{X : U_0 \quad Y : U_0 \quad f : X \to Y}{\surd Poset\ X\ Y\ f : U_1}\ (\surd Poset\ F)\ ,$$

asserts that $\surd Poset\ X\ Y\ f$ is an inhabitant of U_1, if X and Y are inhabitants of U_0, and f is a function that takes X to Y.

Introduction Rules

There are three $\surd Poset$ introduction rules: *base*, *step* and *join*, as depicted in Figure 6.4. The *base* rule (left), which is given by

$$\frac{\begin{array}{lll} X : U_0 & X' : U_0 & \\ Y : U_0 & Y' : U_0 & \surd dl : \surd DataLink\ X\ Y\ f\ X'\ Y'\ f' \\ f : X \to Y & f' : X' \to Y' & \end{array}}{@^1_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl : \surd Poset\ X\ Y\ f}\ (\surd Poset\ I_1)\ ,$$

asserts that $(@^1_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ )$ is a certified data link poset on X, Y and f, if $\surd dl$ is a certified data link component on X, Y, f, X', Y' and f'. For example, if dl_{AW} is the certified data link component as defined in Section 6.2.3, then the certified data link poset $\surd p_{AW}$ on A, W and f_A is given by

$$\frac{\begin{array}{lll} A : U_0 & B : U_0 & \\ W : U_0 & X : U_0 & \surd dl_{AW} : \surd DataLink\ A\ W\ f_A\ B\ X\ f_B \\ f_A : A \to W & f_B : B \to X & \end{array}}{(\surd p_{AW}\ =_{df}\ @^1_{\surd Poset}\ A\ W\ f_A\ B\ X\ f_B\ \surd dl_{AW}\ ) : \surd Poset\ A\ W\ f_A}\ (\surd Poset\ I_1)\ .$$

The *step* rule (centre), which is given by

$$\frac{\begin{array}{lll} X : U_0 & X' : U_0 & \surd dl : \surd DataLink\ X\ Y\ f\ X'\ Y'\ f' \\ Y : U_0 & Y' : U_0 & \surd p' : \surd Poset\ X'\ Y'\ f' \\ f : X \to Y & f' : X' \to Y' & \end{array}}{@^2_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ \surd p' : \surd Poset\ X\ Y\ f}\ (\surd Poset\ I_2)\ ,$$

asserts that $(@^2_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ \surd p'\ )$ is a certified data link poset on X, Y and f, if $\surd dl$ is a certified data link component on X, Y, f, X', Y' and f', and $\surd p'$ is a certified data link poset on X', Y' and f'. Finally, the *join* rule (right), which is given by

$$\frac{\begin{array}{lll} X : U_0 & Y : U_0 & f : X \to Y \end{array} \quad \begin{array}{l} \surd p_1 : \surd Poset\ X\ Y\ f \\ \surd p_2 : \surd Poset\ X\ Y\ f \end{array}}{@^3_{\surd Poset}\ X\ Y\ f\ \surd p_1\ \surd p_2 : \surd Poset\ X\ Y\ f}\ (\surd Poset\ I_3)\ ,$$

asserts that $(@^3_{\surd Poset}\ X\ Y\ f\ \surd p_1\ \surd p_2\ )$ is a certified data link poset on X, Y and f, if $\surd p_1$ and $\surd p_2$ are certified data link posets on X, Y and f.

Elimination Rules

The $\surd Poset$ elimination rules are given by

$$\frac{\begin{array}{l} P\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall f\colon X \to Y\,.\,(\surd Poset\ X\ Y\ f \to U_n) \\ X\colon U_0 \\ Y\colon U_0 \\ f\colon X \to Y \\ \surd p : \surd Poset\ X\ Y\ f \\ i\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall f\colon X \to Y\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,\forall f'\colon X' \to Y'\,. \\ \qquad \forall \surd dl : \surd DataLink\ X\ Y\ f\ X'\ Y'\ f'\,. \\ \qquad\qquad P\ X\ Y\ f\ (@^1_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\,) \\ j\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall f\colon X \to Y\,.\,\forall X'\colon U_0\,.\,\forall Y'\colon U_0\,.\,\forall f'\colon X' \to Y'\,. \\ \qquad \forall \surd dl : \surd DataLink\ X\ Y\ f\ X'\ Y'\ f'\,. \\ \qquad\qquad \forall \surd p' : \surd Poset\ X'\ Y'\ f'\,.\,(P\ X'\ Y'\ f'\ \surd p' \to \\ \qquad\qquad\qquad P\ X\ Y\ f\ (@^2_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ \surd p')) \\ k\colon \forall X\colon U_0\,.\,\forall Y\colon U_0\,.\,\forall f\colon X \to Y\,.\,\forall \surd p_1 : \surd Poset\ X\ Y\ f\,. \\ \qquad \forall \surd p_2 : \surd Poset\ X\ Y\ f\,.\,(P\ X\ Y\ f\ \surd p_1 \to P\ X\ Y\ f\ \surd p_2 \to \\ \qquad\qquad P\ X\ Y\ f\ (@^3_{\surd Poset}\ X\ Y\ f\ \surd p_1\ \surd p_2)) \end{array}}{@^{-1}_{\surd Poset}\ P\ X\ Y\ f\ \surd p\ i\ j\ k\colon P\ X\ Y\ f \surd p}\ (\surd Poset\ E_n)\,,$$

for $n = 0, 1, 2, \ldots$. Each rule asserts, for a particular universe U_n, that $P\ X\ Y\ f \surd p$ is inhabited for all certified data link posets $\surd p$, if

- $P\ X\ Y\ f\ (@^1_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\,)$ is inhabited for all certified data link components $\surd dl$;
- $P X\ Y\ f\ (@^2_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ \surd p'\,)$ is inhabited for all certified data link components $\surd dl$, and all certified data link posets $\surd p'$, if $P\ X'\ Y'\ f'\ \surd p'$ is inhabited for all certified data link posets $\surd p'$; and
- $P\ X\ Y\ f\ (@^3_{\surd Poset}\ X\ Y\ f\ \surd p_1\ \surd p_2\,)$ is inhabited for all certified data link posets $\surd p_1$ and $\surd p_2$, if $P\ X\ Y\ f\ \surd p_1$ is inhabited for all certified data link posets $\surd p_1$, and $P\ X\ Y\ f\ \surd p_2$ is inhabited for all certified data link posets $\surd p_2$.

Computation Rules

The $\surd Poset$ computation rules are given by

$$\begin{array}{l} @^{-1}_{\surd Poset}\ P\ X\ Y\ f\ (@^1_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\,)\ i\ j\ k \twoheadrightarrow i\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl \\ @^{-1}_{\surd Poset}\ P\ X\ Y\ f\ (@^2_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ \surd p'\,)\ i\ j\ k \twoheadrightarrow \\ \qquad j\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ \surd p'\ (@^{-1}_{\surd Poset}\ P\ X'\ Y'\ f'\ \surd p'\ i\ j\ k) \\ @^{-1}_{\surd Poset}\ P\ X\ Y\ f\ (@^3_{\surd Poset}\ X\ Y\ f\ \surd p_1\ \surd p_2\,)\ i\ j\ k \twoheadrightarrow \\ \qquad k\ X\ Y\ f\ \surd p_1\ \surd p_2\ (@^{-1}_{\surd Poset}\ P\ X\ Y\ f\ \surd p_1\ i\ j\ k)\ (@^{-1}_{\surd Poset}\ P\ X\ Y\ f\ \surd p_2\ i\ j\ k)\,. \end{array}$$

Note that the second and third rules are recursive.

Coq Listing 64. *($\surd$Poset)*

```
Inductive CertPoset : forall X : Set, forall Y : Set, (X -> Y) -> Type :=
   Build_CertPoset_1 :
      forall X : Set,
      forall Y : Set,
      forall f : X -> Y,
      forall X' : Set,
      forall Y' : Set,
      forall f' : X' -> Y',
      CertDataLink X Y f X' Y' f' ->
         CertPoset X Y f |
   Build_CertPoset_2 :
      forall X : Set,
      forall Y : Set,
      forall f : X -> Y,
      forall X' : Set,
      forall Y' : Set,
      forall f' : X' -> Y',
      CertDataLink X Y f X' Y' f' ->
      CertPoset X' Y' f' ->
         CertPoset X Y f |
   Build_CertPoset_3 :
      forall X : Set,
      forall Y : Set,
      forall f : X -> Y,
      CertPoset X Y f ->
      CertPoset X Y f ->
         CertPoset X Y f.
```

6.2.5 Certified Ordered Model Transformation

A certified ordered model transformation is either a solitary certified data component, or the composition of a certified data component and a certified data link poset. Figure 6.5 shows two certified ordered model transformations on X, Y and f, comprising on the left, a

$$X \vdash - \underset{\surd d}{\overset{f}{-}} - \succ Y \qquad\qquad X \vdash - \underset{\surd d}{\overset{f}{-}} - \succ Y \;\Big\downarrow \surd p \Big\downarrow$$

Figure 6.5: Two certified ordered model transformations on X, Y and f.

certified data component $\surd d$ on X, Y and f; and on the right, a certified data component $\surd d$ on X, Y and f, and a certified data link poset $\surd p$ on X, Y and f.

Definition 64. *(Certified Ordered Model Transformation)* A certified ordered model transformation on a source class X, a target class Y, and a function f that takes X to Y, is an inhabitant of type $\surd Tran\ X\ Y\ f$.

Formation Rule

The $\surd Tran$ formation rule, which is given by

$$\frac{X : U_0 \quad Y : U_0 \quad f : X \to Y}{\surd Tran\ X\ Y\ f : U_1}\ (\surd Tran\ F)\ ,$$

asserts that $\surd Tran\ X\ Y\ f$ is an inhabitant of U_1, if X and Y are inhabitants of U_0, and f is a function that takes X to Y.

Introduction Rules

There are two $\surd Tran$ introduction rules, as depicted in Figure 6.5. The first rule (on the left), which is given by

$$\frac{X : U_0 \quad Y : U_0 \quad f : X \to Y \quad \surd d : \surd Data\ X\ Y\ f}{@^1_{\surd Tran}\ X\ Y\ f\ \surd d : \surd Tran\ X\ Y\ f}\ (\surd Tran\ I_1)\ ,$$

asserts that $(@^1_{\surd Tran}\ X\ Y\ f\ \surd d\,)$ is a certified ordered model transformation on X, Y and f, if $\surd d$ is a certified data component on X, Y and f. The second rule (on the right), which is given by

$$\frac{\begin{array}{l} X : U_0 \quad Y : U_0 \quad f : X \to Y \quad \surd d : \surd Data\ X\ Y\ f \\ \qquad\qquad\qquad\qquad\qquad\quad \surd p : \surd Poset\ X\ Y\ f \end{array}}{@^2_{\surd Tran}\ X\ Y\ f\ \surd d\ \surd p : \surd Tran\ X\ Y\ f}\ (\surd Tran\ I_2)\ ,$$

asserts that $(@^2_{\surd Tran}\ X\ Y\ f\ \surd d\ \surd p\,)$ is a certified ordered model transformation on X, Y and f, if $\surd d$ is a certified data component on X, Y and f, and $\surd p$ is a certified data link poset on X, Y and f.

Elimination Rules

The $\surd Tran$ elimination rules are given by

$$\frac{\begin{array}{l} P : \forall X : U_0\,.\,\forall Y : U_0\,.\,\forall f : X \to Y\,.\,(\surd Tran\ X\ Y\ f \to U_n) \\ X : U_0 \\ Y : U_0 \\ f : X \to Y \\ \surd t : \surd Tran\ X\ Y \\ i : \forall X : U_0\,.\,\forall Y : U_0\,.\,\forall f : X \to Y\,.\,\forall \surd d : \surd Data\ X\ Y\ f\,. \\ \qquad P\ X\ Y\ f\ (@^1_{\surd Tran}\ X\ Y\ f\ \surd d\,) \\ j : \forall X : U_0\,.\,\forall Y : U_0\,.\,\forall f : X \to Y\,.\,\forall \surd d : \surd Data\ X\ Y\ f\,. \\ \qquad \forall \surd p : \surd Poset\ X\ Y\ f\,.\,P\ X\ Y\ f\ (@^2_{\surd Tran}\ X\ Y\ \surd d\ \surd p\,) \end{array}}{@^{-1}_{\surd Tran}\ P\ X\ Y\ f\ \surd t\ i\ j : P\ X\ Y\ f\ \surd t}\ (\surd Tran\ E_n)\ ,$$

for $n = 0, 1, 2, \ldots$. Each rule asserts, for a particular universe U_n, that $P\ X\ Y\ f\ \surd t$ is inhabited for all certified ordered model transformations $\surd t$, if

- $P\ X\ Y\ f\ (@^1_{\surd Tran}\ X\ Y\ f\ \surd d\,)$ is inhabited for all certified data link components $\surd d$; and
- $P X\ Y\ f\ (@^2_{\surd Tran}\ X\ Y\ \surd d\ \surd p\,)$ is inhabited for all certified data link components $\surd d$, and all certified data link posets $\surd p$.

Computation Rules

The $\surd Tran$ computation rules are given by

$$@^{-1}_{\surd Tran}\ P\ X\ Y\ f\ (@^1_{\surd Tran}\ X\ Y\ f\ \surd d\,)\ i\ j \twoheadrightarrow i\ X\ Y\ f\ \surd d$$
$$@^{-1}_{\surd Tran}\ P\ X\ Y\ f\ (@^2_{\surd Tran}\ X\ Y\ f\ \surd d\ \surd p\,)\ i\ j \twoheadrightarrow j\ X\ Y\ f\ \surd d\ \surd p\ .$$

Coq Listing 65. *($\surd Tran$)*

```
Inductive CertTran: forall X : Set, forall Y : Set, (X -> Y) -> Type :=
   Build_CertTran_1 :
      forall X : Set,
      forall Y : Set,
      forall f : X -> Y,
      CertData X Y f ->
         CertTran X Y f |
   Build_CertTran_2 :
      forall X : Set,
      forall Y : Set,
      forall f : X -> Y,
      CertData X Y f ->
      CertPoset X Y f ->
         CertTran X Y f.
```

6.3 Logical Interpretation Theorem

If $\surd t$ is a certified ordered model transformation on X, Y and f, the logical interpretation of the ordered model transformation that $\surd t$ purports to certify is given by

$$Log_{Tran}\ X\ Y\ (Extract_{Tran}\ X\ Y\ f\ \surd t\,)\ , \tag{6.3}$$

where Log_{Tran} is a function that takes an ordered model transformation to its logical interpretation, as defined in Section 5.3.2, and $Extract_{Tran}$ is a yet-to-be-defined function that extracts an ordered model transformation from a certified ordered model transformation. Informally, the logical interpretation theorem—the focus of this entire section—asserts that (6.3) is provable no matter what the value of $\surd t$. Clearly, not all ordered model transformations are provable. However, those that *are*—according to the logical interpretation theorem—are precisely those that can be extracted from a certified ordered model transformation. The ability to *prove* the logical interpretation of an ordered model transformation t is therefore dependent on the ability to *construct* a certified ordered model transformation

that contains t. To sloganize: the construction of one thing implies the truth of the other. To appreciate why this is the case, one only has to reflect on the kinds of components from which a certified ordered model transformation is ultimately constructed, i.e. certified data and link components. Although it does not necessarily follow that summing the proofs of the parts (the Skolomised forms of the logical interpretations of the underlying data and link components) yields a proof of the whole (the logical interpretation of the entire ordered model transformation), the regular structure of an ordered model transformation ensures that in this case it does.

The proof of the logical interpretation theorem (see Figure 6.6) comprises one theorem and five lemmas, i.e. six *goals* in Coq parlance. Each goal makes an assertion about a particular class of certified ordered model transformations or data link posets as follows.

- Theorem 4 makes an assertion about *all* certified ordered model transformations.
- Lemmas 5 and Lemma 6 make assertions about the subsets of certified ordered model transformations that $@^1_{\surd Tran}$ and $@^2_{\surd Tran}$ construct.
- Within the subset of ordered model transformations that $@^2_{\surd Tran}$ constructs, Lemmas 7, 8 and 9 make assertions about the subsets of data link posets that $@^1_{\surd Poset}$, $@^2_{\surd Poset}$ and $@^3_{\surd Poset}$ construct.

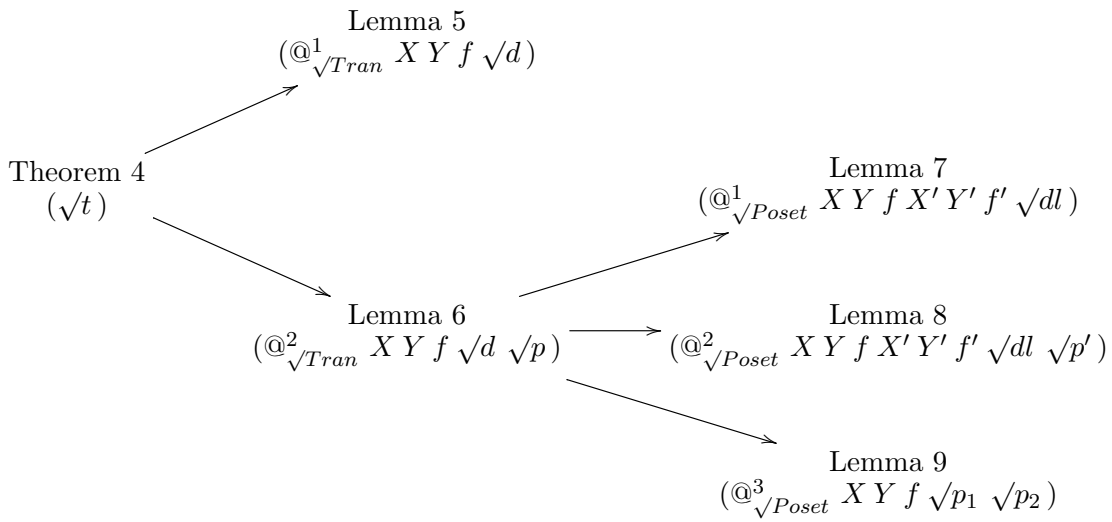

Figure 6.6: The structure of the proof of the logical interpretation theorem.

In the proof of the logical interpretation theorem below, each goal is proved twice: once by hand and then again in Coq. Although the Coq proofs were developed before the corresponding hand crafted proofs, the hand crafted proofs are considered to be the masters, for the author was at all times motivated by the desire to develop generic rather than specific proofs, to ensure that they could easily be ported to other interactive theorem provers if necessary. It is only fair to say that the author would not have succeeded in this

task had it not been for the remarkable Coq Proof Assistant [16, 132]. If the reader wishes to compare and contrast proofs, recall that Coq applies backward reasoning.

The challenge in crafting the proof by hand was not so much in coming up with the proof in the first place, but in finding a suitable means of presenting it. To this end, two sets of functions (T and P) were defined to enhance the readability of the proof.

- The T functions (see Section 6.3.2) define the logical interpretation of an ordered model transformation, as extracted from a particular kind of certified ordered model transformation;
- The P functions (see Section 6.3.3) do likewise for data link posets.

The rest of this section is organised as follows. First, the *Extract* functions are defined: these functions are responsible for extracting uncertified components from certified components. Second, the T and P functions are defined. Third, the six goals are specified and proved.

6.3.1 *Extract* Functions

Each of the following functions extracts an uncertified component from a certified component.

Definition 65. *($Extract_{Data}$)* If $\surd d$ is a certified data component on X, Y, and f, the data component that $\surd d$ certifies is given by

$$Extract_{Data}\, X\, Y\, f\, \surd d\ ,$$

where

$$\begin{aligned} Extract_{Data} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda \surd d\,.\,@^{-1}_{\surd Data}\ P\ X\ Y\ f\ \surd d\ i \\ P &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda \surd d\,.\,Data\ X\ Y \\ i &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda d\,.\,\lambda s\,.\,d\ . \end{aligned}$$

$@^{-1}_{\surd Data}$ is defined in Section 6.2.1.

Example 24. *($Extract_{Data}$)* The data component certified by $\surd d_{AW}$ is given by

$$\begin{aligned} Extract_{Data}\ A\ W\ f_A\ \surd d_{AW} &\twoheadrightarrow @^{-1}_{\surd Data}\ P\ A\ W\ f_A\ (@_{\surd Data}\ A\ W\ f_A\ d_{AW}\ s_{AW})\ i \\ &\twoheadrightarrow i\ A\ W\ f_A\ d_{AW}\ s_{AW} \\ &\twoheadrightarrow d_{AW}\ . \end{aligned}$$

Coq Listing 66. *($Extract_{Data}$)*

```
Definition ExtractData (X : Set) (Y : Set) (f : X -> Y)
   (dp : CertData X Y f) : Data X Y :=

   match dp with
      Build_CertData X Y f d s => d
   end.
```

Definition 66. *($Extract_{Link}$)* If $\surd l$ is a certified link component on X, Y, f, X', Y' and f', the link component that $\surd l$ certifies is given by

$$Extract_{Link}\ X\ Y\ f\ X'\ Y'\ f'\ \surd l\ ,$$

where

$$\begin{aligned} Extract_{Link} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda \surd l\,.\,@^{-1}_{\surd Link}\ P\ X\ Y\ f\ X'\ Y'\ f'\ \surd l\ i \\ P &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda \surd l\,.\,Link\ X\ Y\ X'\ Y' \\ i &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda l\,.\,\lambda c\,.\,l\ . \end{aligned}$$

$@^{-1}_{\surd Link}$ is defined in Section 6.2.2.

Example 25. *($Extract_{Link}$)* The link component certified by $\surd l_{AW}$ is given by

$$\begin{aligned} & Extract_{Link}\ A\ W\ f_A\ B\ X\ f_B\ \surd l_{AW} \\ & \twoheadrightarrow @^{-1}_{\surd Link}\ P\ A\ W\ f_A\ B\ X\ f_B\ (@_{\surd Link}\ A\ W\ f_A\ B\ X\ f_B\ l_{AW}\ c_{AW})\ i \\ & \twoheadrightarrow i\ A\ W\ f_A\ B\ X\ f_B\ l_{AW}\ c_{AW} \\ & \twoheadrightarrow l_{AW}\ . \end{aligned}$$

Coq Listing 67. *($Extract_{Link}$)*

```
Definition ExtractLink (X : Set) (Y : Set) (f : X -> Y)
   (X' : Set) (Y' : Set) (f' : X' -> Y')
   (cl : CertLink X Y f X' Y' f') : Link X Y X' Y' :=

   match cl with
      Build_CertLink X Y f X' Y' f' l c => l
   end.
```

Definition 67. *($Extract_{DataLink}$)* If $\surd dl$ is a certified data link component on X, Y, f, X', Y' and f', the data link component that $\surd dl$ certifies is given by

$$Extract_{DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ ,$$

where

$$\begin{aligned} Extract_{DataLink} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda \surd dl\,.\,@^{-1}_{\surd D'Link}\ P\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ i \\ P &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda \surd dl\,.\,DataLink\ X\ Y\ X'\ Y' \\ i &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda \surd d'\,.\,\lambda \surd l\,.\,@_{DataLink}\ X\ Y\ X'\ Y' \\ & \qquad (Extract_{Data}\ X'\ Y'\ f'\ \surd d')\ (Extract_{Link}\ X\ Y\ f\ X'\ Y'\ f'\ \surd l)\ . \end{aligned}$$

$@^{-1}_{\surd DataLink}$ is defined in Section 6.2.3.

Example 26. *($Extract_{DataLink}$)* The data link component certified by $\surd dl_{AW}$ is given by

$$
\begin{aligned}
&Extract_{DataLink}\ A\ W\ f_A\ B\ X\ f_B\ \surd dl_{AW} \\
&\twoheadrightarrow @^{-1}_{\surd DataLink}\ P\ A\ W\ f_A\ B\ X\ f_B\ (@_{\surd DataLink}\ A\ W\ f_A\ B\ X\ f_B\ \surd d_{BX}\ \surd l_{AW}\ )\ i \\
&\twoheadrightarrow i\ A\ W\ f_A\ B\ X\ f_B\ \surd d_{BX}\ \surd l_{AW} \\
&\twoheadrightarrow @_{DataLink}\ A\ W\ B\ X\ (Extract_{Data}\ B\ X\ f_B\ \surd d_{BX}\ )\ (Extract_{Link}\ A\ W\ f_A\ B\ X\ f_B\ \surd l_{AW}\ ) \\
&\twoheadrightarrow @_{DataLink}\ A\ W\ B\ X\ d_{BX}\ l_{AW} \\
&=_{df}\ dl_{AW}\ .
\end{aligned}
$$

Coq Listing 68. *($Extract_{DataLink}$)*

```
Definition ExtractDataLink (X : Set) (Y : Set) (f : X -> Y)
   (X' : Set) (Y' : Set) (f' : X' -> Y')
   (cdl : CertDataLink X Y f X' Y' f') : DataLink X Y X' Y' :=

   match cdl with
      Build_CertDataLink X Y f X' Y' f' cd' cl =>
         Build_DataLink X Y X' Y'
            (ExtractData X' Y' f' cd')
            (ExtractLink X Y f X' Y' f' cl)
   end.
```

Definition 68. *($Extract_{Poset}$)* If $\surd p$ is a certified data link poset on X, Y, and f, the data link poset that $\surd p$ certifies is given by

$$Extract_{Poset}\ X\ Y\ f\ \surd p\ ,$$

where

$$
\begin{aligned}
Extract_{Poset} &=_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda \surd p\,.\,@^{-1}_{\surd Poset}\ P\ X\ Y\ f\ \surd p\ i\ j\ k \\
P &=_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda \surd p\,.\,Poset\ X\ Y \\
i &=_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda \surd dl\,.\,@^1_{Poset}\ X\ Y\ X'\ Y' \\
&\qquad (Extract_{DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ ) \\
j &=_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda \surd dl\,.\,\lambda \surd p'\,.\,\lambda h'\,.\,@^2_{Poset}\ X\ Y\ X'\ Y' \\
&\qquad (Extract_{DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ )\ (Extract_{Poset}\ X'\ Y'\ f'\ \surd p'\ ) \\
k &=_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda \surd p_1\,.\,\lambda \surd p_2\,.\,\lambda h_1\,.\,\lambda h_2\,.\,@^3_{Poset}\ X\ Y \\
&\qquad (Extract_{Poset}\ X\ Y\ f\ \surd p_1\ )\ (Extract_{Poset}\ X\ Y\ f\ \surd p_2\ )\ .
\end{aligned}
$$

$@^{-1}_{\surd Poset}$ is defined in Section 6.2.4.

Example 27. *($Extract_{Poset}$)* The data link poset certified by $\surd p_{AW}$ is given by

$$
\begin{aligned}
&Extract_{Poset}\ A\ W\ f_A\ \surd p_{AW} \\
&\twoheadrightarrow @^{-1}_{\surd Poset}\ P\ A\ W\ f_A\ (@^2_{\surd Poset}\ A\ W\ f_A\ B\ X\ f_B\ \surd dl_{AW}\ \surd p^3_{BX}\ )\ i\ j\ k
\end{aligned}
$$

$$\begin{aligned}
&\twoheadrightarrow j\ A\ W\ f_A\ B\ X\ f_B\ \surd dl_{AW}\ \surd p^3_{BX}\ (@^{-1}_{\surd Poset}\ P\ B\ X\ f_B\ \surd p^3_{BX}\ i\ j\ k)\\
&\twoheadrightarrow @^2_{Poset}\ A\ W\ B\ X\ (Extract_{DataLink}\ A\ W\ f_A\ B\ X\ f_B\ \surd dl_{AW}\ )\\
&\qquad\qquad (Extract_{Poset}\ B\ X\ f_B\ \surd p^3_{BX}\ )\\
&\twoheadrightarrow @^2_{Poset}\ A\ W\ B\ X\ d_{AW}\ (Extract_{Poset}\ B\ X\ f_B\ \surd p^3_{BX}\ )\ .
\end{aligned}$$

Now, it can be shown that

$$Extract_{Poset}\ B\ X\ f_B\ \surd p^3_{BX}\ \twoheadrightarrow p^3_{BX}\ .$$

$\therefore$

$$\begin{aligned}
Extract_{Poset}\ A\ W\ f_A\ \surd p_{AW}\ &\twoheadrightarrow @^2_{Poset}\ A\ W\ B\ X\ d_{AW}\ p^3_{BX}\\
&=_{df}\ p_{AW}\ .
\end{aligned}$$

Coq Listing 69. *($Extract_{Poset}$)*

```
Fixpoint ExtractPoset (X : Set) (Y : Set) (f : X -> Y)
   (cp : CertPoset X Y f) : Poset X Y :=

   match cp with
      Build_CertPoset_1 X Y f X' Y' f' cdl =>
         Build_Poset_1 X Y X' Y'
            (ExtractDataLink X Y f X' Y' f' cdl) |
      Build_CertPoset_2 X Y f X' Y' f' cdl cp' =>
         Build_Poset_2 X Y X' Y'
            (ExtractDataLink X Y f X' Y' f' cdl)
            (ExtractPoset X' Y' f' cp') |
      Build_CertPoset_3 X Y f cp1 cp2 =>
         Build_Poset_3 X Y
            (ExtractPoset X Y f cp1)
            (ExtractPoset X Y f cp2)
   end.
```

Definition 69. *($Extract_{Tran}$)* If $\surd t$ is a certified ordered model transformation on X, Y, and f, the ordered model transformation that $\surd t$ certifies is given by

$$Extract_{Tran}\ X\ Y\ f\ \surd t\ ,$$

where

$$\begin{aligned}
Extract_{Tran}\ &=_{df}\ \lambda X\ .\ \lambda Y\ .\ \lambda f\ .\ \lambda \surd t\ .\ @^{-1}_{\surd Tran}\ P\ X\ Y\ f\ \surd t\ i\ j\\
P\ &=_{df}\ \lambda X\ .\ \lambda Y\ .\ \lambda f\ .\ \lambda \surd t\ .\ Tran\ X\ Y\\
i\ &=_{df}\ \lambda X\ .\ \lambda Y\ .\ \lambda f\ .\ \lambda \surd d\ .\ @^1_{Tran}\ X\ Y\ (Extract_{Data}\ X\ Y\ f\ \surd d\,)\\
j\ &=_{df}\ \lambda X\ .\ \lambda Y\ .\ \lambda f\ .\ \lambda \surd d\ .\ \lambda \surd p\ .\ @^2_{Tran}\ X\ Y\\
&\qquad (Extract_{Data}\ X\ Y\ f\ \surd d\,)\ (Extract_{Poset}\ X\ Y\ f\ \surd p\,)\ .
\end{aligned}$$

$@^{-1}_{\surd Tran}$ is defined in Section 6.2.5.

Example 28. *($Extract_{Tran}$)* The ordered model transformation certified by $\surd t^2_{AW}$ is given by

$$
\begin{aligned}
& Extract_{Tran}\, A\, W\, f_A\, \surd t^2_{AW} \\
& \twoheadrightarrow @^{-1}_{\surd Tran}\, P\, A\, W\, f_A\, (@^2_{\surd Tran}\, A\, W\, f_A\, \surd d_{AW}\ \surd p_{AW}\,)\, i\, j \\
& \twoheadrightarrow j\, A\, W\, f_A\, \surd d_{AW}\ \surd p_{AW} \\
& \twoheadrightarrow @^2_{Tran}\, A\, W\, (Extract_{Data}\, A\, W\, f_A\, \surd d_{AW}\,)\, (Extract_{Poset}\, A\, W\, f_A\, \surd p_{AW}\,) \\
& \twoheadrightarrow @^2_{Tran}\, A\, W\, d_{AW}\, p_{AW} \\
& =_{df}\ t^2_{AW}\ .
\end{aligned}
$$

Coq Listing 70. *($Extract_{Tran}$)*

```
Definition ExtractTran (X : Set) (Y : Set) (f : X -> Y)
   (ct : CertTran X Y f) : Tran X Y :=

   match ct with |
      Build_CertTran_1 X Y f cd =>
         Build_Tran_1 X Y (ExtractData X Y f cd) |
      Build_CertTran_2 X Y f cd cp =>
         Build_Tran_2 X Y
            (ExtractData X Y f cd)
            (ExtractPoset X Y f cp)
   end.
```

6.3.2 T Functions

The sole purpose of the T functions, and their allies the P functions, is to enhance the readability of the proof of the logical interpretation theorem. The T functions comprise a principal function $\widehat{T}$, and six subordinate functions T_{11} through T_{23}, all of which directly or indirectly call $\widehat{T}$, as shown in Figure 6.7.

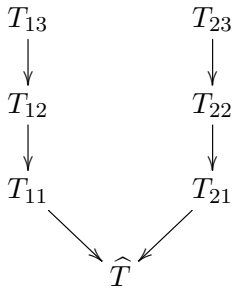

Figure 6.7: The T functions. The functions in each branch are tied to the same constructor, e.g. T_{11}, T_{12} and T_{13} are tied to $@^1_{\surd Tran}$.

The principal function, which is given by

$$\widehat{T}\ =_{df}\ \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda \surd t\ .\, Log_{Tran}\, X\, Y\, (Extract_{Tran}\, X\, Y\, f\, \surd t\,)\ ,$$

takes a certified ordered model transformation $\surd t$, to the logical interpretation of the ordered model transformation that $\surd t$ certifies. For example,

$$\begin{aligned}\widehat{T}\ A\ W\ f_A\ \surd t^1_{AW} &\twoheadrightarrow Log_{Tran}\ A\ W\ t^1_{AW}\\ &\twoheadrightarrow \forall a\colon A\,.\,Pre_A\ a \to \exists w\colon W\,.\,Post_W\ a\ w\ .\end{aligned}$$

The subordinate functions, which are given by

$$\begin{aligned}T_{11} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda \surd d\ .\\ &\qquad \widehat{T}\ X\ Y\ f\ (@^1_{\surd Tran}\ X\ Y\ f\ \surd d\,)\\ T_{12} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda d\,.\,\lambda skol\,.\\ &\qquad T_{11}\ X\ Y\ f\ (@_{\surd Data}\ X\ Y\ f\ d\ skol)\\ T_{13} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda Pre\,.\,\lambda Post\,.\,\lambda skol\,.\\ &\qquad T_{12}\ X\ Y\ f\ (@_{Data}\ X\ Y\ Pre\ Post)\ skol\ ,\end{aligned}$$

and

$$\begin{aligned}T_{21} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda \surd d\ .\,\lambda \surd p\ .\\ &\qquad \widehat{T}\ X\ Y\ f\ (@^2_{\surd Tran}\ X\ Y\ f\ \surd d\ \surd p\,)\\ T_{22} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda d\,.\,\lambda skol\,.\,\lambda \surd p\ .\\ &\qquad T_{21}\ X\ Y\ f\ (@_{\surd Data}\ X\ Y\ f\ d\ skol)\ \surd p\\ T_{23} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda Pre\,.\,\lambda Post\,.\,\lambda skol\,.\,\lambda \surd p\ .\\ &\qquad T_{22}\ X\ Y\ f\ (@_{Data}\ X\ Y\ Pre\ Post)\ skol\ \surd p\ ,\end{aligned}$$

also take a certified ordered model transformation to the logical interpretation of an ordered model transformation. In doing so, they account for every kind of ordered model transformation that can be passed to $\widehat{T}$. For instance, T_{11} passes

$$@^1_{\surd Tran}\ X\ Y\ f\ \surd d\ ,$$

T_{12} passes the more specific

$$@^1_{\surd Tran}\ X\ Y\ f\ (@_{\surd Data}\ X\ Y\ f\ d\ skol)\ ,$$

and T_{13} passes the even more specific

$$@^1_{\surd Tran}\ X\ Y\ f\ (@_{\surd Data}\ X\ Y\ f\ (@_{Data}\ X\ Y\ Pre\ Post)\ skol)\ .$$

Clearly, the specificity of the ordered model transformation that T_{ij} passes to $\widehat{T}$, increases with j, where i denotes the means by which it is constructed, i.e. $@^i_{\surd Tran}$.

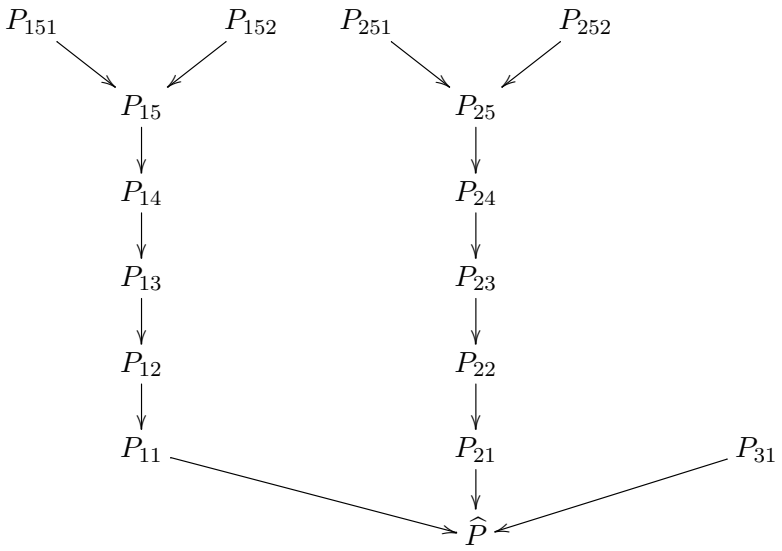

Figure 6.8: The P functions. The functions in each main branch are associated with the same constructor.

6.3.3 P Functions

The P functions comprise a principal function $\widehat{P}$, and fifteen subordinate functions P_{11} though P_{31}, all of which directly or indirectly call $\widehat{P}$, as shown in Figure 6.8.

The principal function, which is given by

$$\widehat{P} =_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda\surd p\,.\,Log_{Poset}\,X\,Y\,(Extract_{Poset}\,X\,Y\,f\,\surd p)\,,$$

takes a certified data link poset $\surd p$, to the logical interpretation of the data link poset that $\surd p$ certifies. The subordinate functions, which are given by

$$\begin{aligned}
P_{11} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda\surd dl\,.\\
&\qquad \widehat{P}\,X\,Y\,f\,(@^1_{\surd Poset}\,X\,Y\,f\,X'\,Y'\,f'\,\surd dl\,)\\
P_{12} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda\surd d'\,.\,\lambda\surd l\,.\\
&\qquad P_{11}\,X\,Y\,f\,X'\,Y'\,f'\,(@_{\surd DataLink}\,X\,Y\,f\,X'\,Y'\,f'\,\surd d'\,\surd l\,)\\
P_{13} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda d'\,.\,\lambda skol'\,.\,\lambda\surd l\,.\\
&\qquad P_{12}\,X\,Y\,f\,X'\,Y'\,f'\,(@_{\surd Data}\,X'\,Y'\,f'\,d'\,skol')\,\surd l\\
P_{14} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda Pre'\,.\,\lambda Post'\,.\,\lambda skol'\,.\,\lambda\surd l\,.\\
&\qquad P_{13}\,X\,Y\,f\,X'\,Y'\,f'\,(@_{Data}\,X'\,Y'\,Pre'\,Post')\,skol'\,\surd l\\
P_{15} &=_{df} \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda Pre'\,.\,\lambda Post'\,.\,\lambda skol'\,.\,\lambda l\,.\,\lambda com\,.\\
&\qquad P_{14}\,X\,Y\,f\,X'\,Y'\,f'\,Pre'\,Post'\,skol'\,(@_{\surd Link}\,X\,Y\,f\,X'\,Y'\,f'\,l\,com)\,,
\end{aligned}$$

and

$$\begin{aligned}
P_{151} =_{df}\ & \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda Pre'\,.\,\lambda Post'\,.\,\lambda skol'\,.\,\lambda r\,.\,\lambda s\,.\,\lambda com\,.\\
& P_{15}\ X\ Y\ f\ X'\ Y'\ f'\ Pre'\ Post'\ skol'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ com\\
P_{152} =_{df}\ & \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda Pre'\,.\,\lambda Post'\,.\,\lambda skol'\,.\,\lambda r\,.\,\lambda s\,.\,\lambda com\,.\\
& P_{15}\ X\ Y\ f\ X'\ Y'\ f'\ Pre'\ Post'\ skol'\ (@^2_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ com\ ,
\end{aligned}$$

and

$$\begin{aligned}
P_{21} =_{df}\ & \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda\surd dl\ .\,\lambda\surd p'\ .\\
& \widehat{P}\ X\ Y\ f\ (@^2_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ \surd p')\\
P_{22} =_{df}\ & \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda\surd d'\ .\,\lambda\surd l\ .\,\lambda\surd p'\ .\\
& P_{21}\ X\ Y\ f\ X'\ Y'\ f'\ (@_{\surd DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd d'\ \surd l\,)\ \surd p'\\
P_{23} =_{df}\ & \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda d'\,.\,\lambda skol'\,.\,\lambda\surd l\ .\,\lambda\surd p'\ .\\
& P_{22}\ X\ Y\ f\ X'\ Y'\ f'\ (@_{\surd Data}\ X'\ Y'\ f'\ d'\ skol')\ \surd l\ \surd p'\\
P_{24} =_{df}\ & \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda Pre'\,.\,\lambda Post'\,.\,\lambda skol'\,.\,\lambda\surd l\ .\,\lambda\surd p'\ .\\
& P_{23}\ X\ Y\ f\ X'\ Y'\ f'\ (@_{Data}\ X'\ Y'\ Pre'\ Post')\ skol'\ \surd l\ \surd p'\\
P_{25} =_{df}\ & \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda Pre'\,.\,\lambda Post'\,.\,\lambda skol'\,.\,\lambda l\,.\,\lambda com\,.\,\lambda\surd p'\ .\\
& P_{24}\ X\ Y\ f\ X'\ Y'\ f'\ Pre'\ Post'\ skol'\ (@_{\surd Link}\ X\ Y\ f\ X'\ Y'\ f'\ l\ com)\ \surd p'\ ,
\end{aligned}$$

and

$$\begin{aligned}
P_{251} =_{df}\ & \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda Pre'\,.\,\lambda Post'\,.\,\lambda skol'\,.\,\lambda r\,.\,\lambda s\,.\,\lambda com\,.\,\lambda\surd p'\ .\\
& P_{25}\ X\ Y\ f\ X'\ Y'\ f'\ Pre'\ Post'\ skol'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ com\ \surd p'\\
P_{252} =_{df}\ & \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda X'\,.\,\lambda Y'\,.\,\lambda f'\,.\,\lambda Pre'\,.\,\lambda Post'\,.\,\lambda skol'\,.\,\lambda r\,.\,\lambda s\,.\,\lambda com\,.\,\lambda\surd p'\ .\\
& P_{25}\ X\ Y\ f\ X'\ Y'\ f'\ Pre'\ Post'\ skol'\ (@^2_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ com\ \surd p'\ ,
\end{aligned}$$

and

$$\begin{aligned}
P_{31} =_{df}\ & \lambda X\,.\,\lambda Y\,.\,\lambda f\,.\,\lambda\surd p_1\ .\,\lambda\surd p_2\ .\\
& \widehat{P}\ X\ Y\ f\ (@^3_{Poset}\ X\ Y\ f\ \surd p_1\ \surd p_2\,)\ ,
\end{aligned}$$

also take a certified data link poset to the logical interpretation of a data link poset. In doing so, they account for every kind of data link poset that can be passed to $\widehat{P}$. For example, P_{11} passes

$$@^1_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ ,$$

P_{12} passes the more specific

$$@^1_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ (@_{\surd DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd d\ \surd l\,)\ ,$$

P_{13} passes the even more specific

$$@^1_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ (@_{\surd DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ (@_{\surd Data}\ X'\ Y'\ f'\ d'\ skol')\ \surd l\,)\ ,$$

and so on. Clearly, the specificity of the data link poset that P_{ij} passes to $\widehat{P}$, increases with j, where i denotes the means by which it is constructed, i.e. $@^i_{\surd Poset}$.

6.3.4 Goals

The six goals, as identified in Section 6.3, are proved in a top down manner (from left to right, top to bottom in Figure 6.6) using forward references to goals that have not yet been proved. Out of necessity, the following conventions apply.

- Very few variables are accompanied by their types. However, given the number of times that X, for example, appears alongside its type U_0 in the foregoing text, this is unlikely to cause confusion. The same applies to other common variables such as Y, f, Pre and $Post$ and their primed variants; and d, l, dl, p, t and their certified variants $\surd d$, $\surd l$, $\surd dl$, $\surd p$, and $\surd t$.

- A set of universal quantifications is more or less reduced to a set of bound variables. For example,
$$\forall X : U_0 \,.\, \forall Y : U_0 \,.\, \forall f : X \to Y$$
is denoted by $\forall_{[X,\,Y,\,f\,]}$.

- If Γ is a context and x is a variable of type X in Γ, then
$$\frac{x : X \in \Gamma}{\Gamma \vdash x : X}\ (Ass)$$
is abbreviated
$$\Gamma \vdash x : X_{(Ass)}\ .$$

- In applications of the $Tran$ and $Poset$ elimination rules, the premisses are deliberately not reduced to a form that would render them easily recognisable as the premisses of an elimination rule for a given conclusion, since this would undermine their *raison d'être*, which is to enhance the readability of the proof. For example, reducing
$$T_{11}\ X\ Y\ f\ \surd d$$
to
$$\widehat{T}\ X\ Y\ f\ (@^1_{\surd Tran}\ X\ Y\ f\ \surd d\,)$$
in the proof of Theorem 4, would not produce a more readable proof. Further (and better) examples can be found in the final steps of Lemmas 7 and 8.

- For pedagogical reasons, the universal quantifications in the premisses of some elimination rules are partitioned into two groups: a mandatory group as defined by the elimination rule, and an optional group as defined by the remaining free variables in the proposition over which the elimination takes place. For example, the first group of bound variables in
$$\vdash \forall_{[X,\,Y,\,Pre,\,Post\,]} \,.\, \forall_{[f,\,s,\,\surd p\,]} \,.\, T_{23}\ X\ Y\ f\ Pre\ Post\ s\ \surd p\ \ ,$$
a premiss in Lemma 6, is mandated by the $(Data\,E_1)$ rule, whereas the second group is defined by the remaining free variables in $T_{23}\ X\ Y\ f\ Pre\ Post\ s\ \surd p$.

The proof of the logical interpretation theorem starts with the proof of the top level goal, i.e. Theorem 4.

Theorem 4. *(Logical Interpretation)* The logical interpretation of an ordered model transformation, as extracted from a certified ordered model transformation, is always provable, i.e.

$$\vdash \forall_{[X,\,Y,\,f,\,\surd t\,]}\,.\,\widehat{T}\;X\;Y\;f\;\surd t\;\;.$$

Proof. Use Lemmas 5 and 6.

$$\frac{\dfrac{\vdash \forall_{[X,\,Y,\,f,\,\surd d\,]}\,.\,T_{11}\;X\;Y\;f\;\surd d \qquad \vdash \forall_{[X,\,Y,\,f,\,\surd d\,,\,\surd p\,]}\,.\,T_{21}\;X\;Y\;f\;\surd t\;\surd p}{[X,\,Y,\,f,\,\surd t\,] \vdash \widehat{T}\;X\;Y\;f\;\surd t}\;(\surd Tran\;E_1)}{\vdash \forall_{[X,\,Y,\,f,\,\surd t\,]}\,.\,\widehat{T}\;X\;Y\;f\;\surd t}\;(\forall I)_*\;.$$

□

Coq Listing 71. *(Theorem 4)*

```
Theorem Theorem_Tran :

forall X : Set,
forall Y : Set,
forall f : X -> Y,
forall ct : CertTran X Y f,
   LogTran X Y (ExtractTran X Y f ct).

Proof.
   intros X Y f ct.
   destruct ct as [X Y f cd | X Y f cd cp].
   exact (Lemma_Tran_1 X Y f cd).
   exact (Lemma_Tran_2 X Y f cd cp).
Qed.
```

Lemma 5. The logical interpretation of an ordered model transformation, as extracted from a certified ordered model transformation of the form

$$@^1_{\surd Tran}\;X\;Y\;f\;\surd d\;\;,$$

is provable for all certified data components $\surd d$, i.e.

$$\vdash \forall_{[X,\,Y,\,f,\,\surd d\,]}\,.\,T_{11}\;X\;Y\;f\;\surd d\;\;. \tag{6.4}$$

Proof. Let

$$\Delta\;=_{df}\;[X,\,Y,\,f,\,Pre,\,Post,\,s\colon Skol\;X\;Y\;(@_{Data}\;X\;Y\;Pre\;Post)\;f\,]\;.$$

1. Reduce

$$Skol\;X\;Y\;(@_{Data}\;X\;Y\;Pre\;Post)\;f\;\;.$$

$$\begin{aligned} \mathit{Skol}\ X\ Y\ (@_{\mathit{Data}}\ X\ Y\ \mathit{Pre}\ \mathit{Post})\ f &\twoheadrightarrow @^{-1}_{\mathit{Data}}\ P\ X\ Y\ (@_{\mathit{Data}}\ X\ Y\ \mathit{Pre}\ \mathit{Post})\ i \\ &\twoheadrightarrow i\ X\ Y\ \mathit{Pre}\ \mathit{Post} \\ &\twoheadrightarrow \forall x\colon X\,.\,\mathit{Pre}\ x \to \mathit{Post}\ x\ (f\ x)\ . \end{aligned}$$

2. Prove

$$\Delta \vdash \forall x\colon X\,.\,\mathit{Pre}\ x \to \exists y\colon Y\,.\,\mathit{Post}\ x\ y\ .$$

Use step 1.

$$\dfrac{\dfrac{\Delta \vdash \mathit{Skol}\ X\ Y\ (@_{\mathit{Data}}\ X\ Y\ \mathit{Pre}\ \mathit{Post})\ f_{(\mathit{Ass})}}{\Delta \vdash \forall x\colon X\,.\,\mathit{Pre}\ x \to \mathit{Post}\ x\ (f\ x)}\ (=_{\mathit{red}}) \quad [X,\,x\,] \vdash x\colon X_{(\mathit{Ass})}}{\dfrac{\dfrac{[X,\,\mathit{Pre},\,x,\,h\colon \mathit{Pre}\ x\,] \vdash \mathit{Pre}\ x_{(\mathit{Ass})}}{\Delta, [x,\,h\,] \vdash \mathit{Post}\ x\ (f\ x)}\ (\forall E)(\to E)}{\Delta, [x,\,h\,] \vdash \exists y\colon Y\,.\,\mathit{Post}\ x\ y}\ (\exists\, I)}{\Delta \vdash \forall x\colon X\,.\,\mathit{Pre}\ x \to \exists y\colon Y\,.\,\mathit{Post}\ x\ y}\ (\to I)\,(\forall I)\ .$$

3. Reduce

$$T_{13}\ X\ Y\ f\ \mathit{Pre}\ \mathit{Post}\ s\ .$$

$$\begin{aligned} T_{13}\ X\ Y\ f\ \mathit{Pre}\ \mathit{Post}\ s &\twoheadrightarrow T_{12}\ X\ Y\ f\ \underbrace{(@_{\mathit{Data}}\ X\ Y\ \mathit{Pre}\ \mathit{Post})}_{d}\ s \\ &\twoheadrightarrow T_{11}\ X\ Y\ f\ \underbrace{(@_{\surd \mathit{Data}}\ X\ Y\ f\ \delta\ s)}_{\surd d} \\ &\twoheadrightarrow \widehat{T}\ X\ Y\ f\ \underbrace{(@^1_{\surd \mathit{Tran}}\ X\ Y\ f\ \surd\delta\,)}_{\surd t} \\ &\twoheadrightarrow \mathit{Log}_{\mathit{Tran}}\ X\ Y\ (\mathit{Extract}_{\mathit{Tran}}\ X\ Y\ f\ \surd t\,) \\ &\twoheadrightarrow \mathit{Log}_{\mathit{Tran}}\ X\ Y\ (@^1_{\mathit{Tran}}\ X\ Y\ (\mathit{Extract}_{\mathit{Data}}\ X\ Y\ f\ \surd d\,)) \\ &\twoheadrightarrow \mathit{Log}_{\mathit{Tran}}\ X\ Y\ (@^1_{\mathit{Tran}}\ X\ Y\ d) \\ &\twoheadrightarrow \mathit{Log}_{\mathit{Data}}\ X\ Y\ d \\ &\twoheadrightarrow \forall x\colon X\,.\,\mathit{Pre}\ x \to \exists y\colon Y\,.\,\mathit{Post}\ x\ y\ . \end{aligned}$$

4. Finally, prove (6.4). Use steps 2 and 3.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta \vdash \forall x\colon X\,.\,\mathit{Pre}\ x \to \exists y\colon Y\,.\,\mathit{Post}\ x\ y}{\Delta \vdash T_{13}\ X\ Y\ f\ \mathit{Pre}\ \mathit{Post}\ s}\ (=_{\mathit{abs}})}{\vdash \forall_{[X,\,Y,\,\mathit{Pre},\,\mathit{Post}\,]}\,.\,\forall_{[f,\,s\,]}\,.\,T_{13}\ X\ Y\ f\ \mathit{Pre}\ \mathit{Post}\ s}\ (\forall I)_*}{[X,\,Y,\,d\,] \vdash \forall_{[f,\,s\,]}\,.\,T_{12}\ X\ Y\ f\ d\ s}\ (\mathit{Data}\ E_1)}{\vdash \forall_{[X,\,Y,\,f,\,d,\,s\,]}\,.\,T_{12}\ X\ Y\ f\ d\ s}\ (\forall I)_*}{[X,\,Y,\,f,\,\surd d\,] \vdash T_{11}\ X\ Y\ f\ \surd d}\ (\surd \mathit{Data}\ E_1)}{\vdash \forall_{[X,\,Y,\,f,\,\surd d\,]}\,.\,T_{11}\ X\ Y\ f\ \surd d}\ (\forall I)_*\ .$$

□

Coq Listing 72. *(Lemma 5)*

```
Lemma Lemma_Tran_1 :

forall X : Set,
forall Y : Set,
forall f : X -> Y,
forall cd : CertData X Y f,
   LogTran X Y (ExtractTran X Y f (Build_CertTran_1 X Y f cd)).

Proof.
   intros X Y f cd.
   destruct cd as [X Y f d skol].
   destruct d as [X Y Pre Post].
   simpl.
   intros.
   exists (f x).
   exact (skol x H).
Qed.
```

Lemma 6. The logical interpretation of an ordered model transformation, as extracted from a certified ordered model transformation of the form

$$@^2_{Tran}\, X\, Y\, f\, \surd d\, \surd p\ ,$$

is provable for all certified data components $\surd d$, and all certified data link posets $\surd p$, i.e.

$$\vdash \forall_{[X,\,Y,\,f,\,\surd d\,,\,\surd p\,]}\,.\,T_{21}\, X\, Y\, f\, \surd d\, \surd p\ . \tag{6.5}$$

Proof. Let

$$\Delta\ =_{df}\ [X,\, Y,\, f,\, Pre,\, Post,\, s\colon Skol\, X\, Y\, (@_{Data}\, X\, Y\, Pre\, Post)\, f\,]\ .$$

1. Prove
$$\Delta, [x\colon X,\, h\colon Pre\, x\,] \vdash Post\, x\, (f\, x)\ .$$
Use step 1 of Lemma 5.
$$\dfrac{\dfrac{\Delta \vdash Skol\, X\, Y\, (@_{Data}\, X\, Y\, Pre\, Post)\, f}{\Delta \vdash \forall x\colon X\,.\, Pre\, x \to Post\, x\, (f\, x)}\ (=_{red})}{\dfrac{\begin{array}{l}[X,\, x\,] \vdash x_{(Ass)} \\ [X,\, Pre,\, x,\, h\colon Pre\, x\,] \vdash Pre\, x_{(Ass)}\end{array}}{\Delta, [x,\, h\,] \vdash Post\, x\, (f\, x)}\ (\forall E)(\to E)}\ .$$

2. Prove

$$[X,\, Y,\, f,\, x\,] \vdash \forall \surd p\,.\, \widehat{P}\; X\; Y\; f\; \surd p\; x\; (f\; x)\;.$$

Use Lemmas 7, 8 and 9.

$$\dfrac{\begin{array}{l}\vdash \forall_{\Gamma_3}\,.\,\forall x\,.\,P_{11}\; X\; Y\; f\; X'\; Y'\; \surd dl\; x\; (f\; x)\\ \vdash \forall_{\Gamma_4}\,.\,(\forall x'\,.\,\widehat{P}\; X'\; Y'\; f'\; p'\; x'\; (f'\; x')) \rightarrow\\ \qquad \forall x\,.\,P_{21}\; X\; Y\; f\; X'\; Y'\; f'\; \surd dl\; p'\; x\; (f\; x)\\ \vdash \forall_{\Gamma_5}\,.\,(\forall x\colon X\,.\,\widehat{P}\; X\; Y\; f\; \surd p_1\; x\; (f\; x)) \rightarrow\\ \qquad (\forall x\,.\,\widehat{P}\; X\; Y\; f\; \surd p_2\; x\; (f\; x)) \rightarrow\\ \qquad \forall x\,.\,P_{31}\; X\; Y\; f\; \surd p_1\; \surd p_2\; x\; (f\; x)\end{array}}{[X,\, Y,\, f,\, \surd p\,] \vdash \forall x\,.\,\widehat{P}\; X\; Y\; f\; \surd p\; x\; (f\; x)}\; (\surd Poset\; E_1)$$

$$\dfrac{\dfrac{[X,\, x\,] \vdash x_{(Ass)}}{[X,\, Y,\, f,\, \surd p\,,\, x\,] \vdash \widehat{P}\; X\; Y\; f\; \surd p\; x\; (f\; x)}\; (\forall E)}{[X,\, Y,\, f,\, x\,] \vdash \forall \surd p\,.\,\widehat{P}\; X\; Y\; f\; \surd p\; x\; (f\; x)}\; (\forall I)\;.$$

3. Prove

$$\Delta \vdash \forall x\colon X\,.\,Pre\; x \rightarrow \exists y\colon Y\,.\,(Post\; x\; y\; \wedge \forall \surd p\,.\,\widehat{P}\; X\; Y\; f\; \surd p\; x\; y)\;.$$

Use steps 1 and 2.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\Delta, [x,\, h\,] \vdash Post\; x\; (f\; x)\\ [X,\, Y,\, f,\, x\,] \vdash \forall \surd p\,.\,\widehat{P}\; X\; Y\; f\; \surd p\; x\; (f\; x)\end{array}}{\Delta, [x,\, h\,] \vdash Post\; x\; (f\; x)\; \wedge \forall \surd p\,.\,\widehat{P}\; X\; Y\; f\; \surd p\; x\; (f\; x)}\; (\wedge I)}{\Delta, [x,\, h\,] \vdash \exists y\colon Y\,.\,(Post\; x\; y\; \wedge \forall \surd p\,.\,\widehat{P}\; X\; Y\; f\; \surd p\; x\; y)}\; (\exists I)}{\Delta \vdash \forall x\colon X\,.\,Pre\; x \rightarrow \exists y\colon Y\,.\,(Post\; x\; y\; \wedge \forall \surd p\,.\,\widehat{P}\; X\; Y\; f\; \surd p\; x\; y)}\; (\rightarrow I)\,(\forall I)\;.$$

4. Convert

$$\forall \surd p\,.\,T_{23}\; X\; Y\; f\; Pre\; Post\; s\; \surd p\;.$$

$$\begin{array}{l} T_{23}\; X\; Y\; f\; Pre\; Post\; s\; \surd p\\ \quad \twoheadrightarrow T_{22}\; X\; Y\; f\; \underbrace{(@_{Data}\; X\; Y\; f\; Pre\; Post)}_{d}\; s\; \surd p\\ \quad \twoheadrightarrow T_{21}\; X\; Y\; f\; \underbrace{(@_{\surd Data}\; X\; Y\; f\; d\; s)}_{\surd d}\; \surd p\\ \quad \twoheadrightarrow \widehat{T}\; X\; Y\; f\; \underbrace{(@^2_{\surd Tran}\; X\; Y\; f\; \surd d\; \surd p\,)}_{\surd t}\\ \quad \twoheadrightarrow Log_{Tran}\; X\; Y\; (Extract_{Tran}\; X\; Y\; f\; \surd t\,)\\ \quad \twoheadrightarrow Log_{Tran}\; X\; Y\; (@^2_{Tran}\; X\; Y\; (Extract_{Data}\; X\; Y\; f\; \surd d\,)\; \underbrace{(Extract_{Poset}\; X\; Y\; f\; \surd p\,)}_{p}) \end{array}$$

$$\twoheadrightarrow Log_{Tran}\ X\ Y\ (@^2_{Tran}\ X\ Y\ d\ p)$$
$$\twoheadrightarrow \forall x\colon X\,.\,Pre\ x \to \exists y\colon Y\,.\,(Post\ x\ y\ \wedge Log_{Poset}\ X\ Y\ p\ x\ y)$$
$$\twoheadleftarrow \forall x\colon X\,.\,Pre\ x \to \exists y\colon Y\,.\,(Post\ x\ y\ \wedge \widehat{P}\ X\ Y\ f\ \surd p\ x\ y)\,.$$

$\therefore$

$$\forall \surd p\,.\,T_{23}\ X\ Y\ f\ Pre\ Post\ s\ \surd p$$
$$\leftrightarrow\!\!\!\rightarrow \forall x\colon X\,.\,Pre\ x \to \exists y\colon Y\,.\,(Post\ x\ y\ \wedge \forall \surd p\,.\,\widehat{P}\ X\ Y\ f\ \surd p\ x\ y)\,.$$

5. Finally, prove (6.5). Use steps 3 and 4.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta \vdash \forall x\colon X\,.\,Pre\ x \to \exists y\colon Y\,.\,(Post\ x\ y\ \wedge \forall \surd p\,.\,\widehat{P}\ X\ Y\ f\ \surd p\ x\ y)}{\Delta \vdash \forall \surd p\,.\,T_{23}\ X\ Y\ f\ Pre\ Post\ s\ \surd p}\ (=_{conv})}{\vdash \forall_{[X,Y,Pre,Post]}\,.\,\forall_{[f,s,\surd p]}\,.\,T_{23}\ X\ Y\ f\ Pre\ Post\ s\ \surd p}\ (\forall I)_*}{[X,\ Y,\ d] \vdash \forall_{[f,s,\surd p]}\,.\,T_{22}\ X\ Y\ f\ d\ s\ \surd p}\ (Data\ E_1)}{\vdash \forall_{[X,Y,f,d,s]}\,.\,\forall \surd p\,.\,T_{22}\ X\ Y\ f\ d\ s\ \surd p}\ (\forall I)_*}{[X,\ Y,\ f,\ \surd d\ ] \vdash \forall \surd p\,.\,T_{21}\ X\ Y\ f\ \surd d\ \surd p}\ (\surd Data\ E_1)}{\vdash \forall_{[X,Y,f,\surd d,\surd p]}\,.\,T_{21}\ X\ Y\ f\ \surd d\ \surd p}\ (\forall I)_*\,.$$

□

Coq Listing 73. *(Lemma 6)*

```
Lemma Lemma_Tran_2 :

forall X : Set,
forall Y : Set,
forall f : X -> Y,
forall cd : CertData X Y f,
forall cp : CertPoset X Y f,
   LogTran X Y (ExtractTran X Y f (Build_CertTran_2 X Y f cd cp)).

Proof.
   intros X Y f cd cp.
   destruct cd as [X Y f d skol].
   destruct d as [X Y Pre Post].
   simpl.
   intros x H.
   exists (f x).
   split.
   exact (skol x H).
   clear H skol Pre Post.
   induction cp as [X Y f X' Y' f' cdl | X Y f X' Y' f' cdl cp' IHcp' |
      X Y f cp1 IHcp1 cp2 IHcp2].
   exact (Lemma_Poset_1 X Y f X' Y' f' cdl x).
```

```
   exact (Lemma_Poset_2 X Y f X' Y' f' cdl cp' IHcp' x).
   exact (Lemma_Poset_3 X Y f cp1 cp2 IHcp1 IHcp2 x).
Qed.
```

Lemma 7. The logical interpretation of a data link poset, as extracted from a certified data link poset of the form

$$@^1_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ ,$$

is provable on all objects x of X and $(f\ x)$ of Y, for all certified data links $\surd dl$, i.e.

$$\vdash \forall_{[X,\,Y,\,f,\,X',\,Y',\,f',\,\surd dl\,]}\,.\,\forall x\colon X\,.\,P_{11}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ x\ (f\ x)\ . \qquad (6.6)$$

Proof. Let

$$\Delta\ =_{df}\ \Delta_1 \cup \Delta_2\ ,$$

where

$$\begin{aligned}\Delta_1\ &=_{df}\ [X',\,Y',\,f',\,Pre',\,Post',\,s'\colon Skol\ X'\ Y'\ (@_{Data}\ X'\ Y'\ Pre'\ Post')\ f'\,]\\ \Delta_2\ &=_{df}\ [X,\,Y,\,f,\,X',\,Y',\,f',\,l,\,c\colon Com\ X\ Y\ X'\ Y'\ l\ f\ f'\,]\ .\end{aligned}$$

Further, let

$$\begin{aligned}\Delta_3\ &=_{df}\ [\,X,\,Y,\,f,\,X',\,Y',\,f',\,r,\,s,\,c\colon Com\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ f\ f'\,]\\ \Delta_4\ &=_{df}\ [X,\,Y,\,X',\,Y',\,r\colon X \to X',\,s\colon Y \to Y'\,]\\ \Delta_5\ &=_{df}\ [\,X,\,Y,\,f,\,X',\,Y',\,f',\,r,\,s,\,c\colon Com\ X\ Y\ X'\ Y'\ (@^2_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ f\ f'\,]\\ \Delta_6\ &=_{df}\ [X,\,Y,\,X',\,Y',\,r\colon X \to [\,X'\,],\,s\colon Y \to [\,Y'\,]\,]\ .\end{aligned}$$

1. Prove
$$\Delta_1, [x',\,h_1\colon Pre'\ x'\,] \vdash Post'\ x'\ (f'\ x')\ .$$
The proof is similar to that of step 1 of Lemma 5, except that it relates to primed terms instead of non-primed terms.

2. Prove
$$\vdash \forall_{\Delta_3,[x,\,x',\,h_2\,]}\,.\,Link_{Post}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ (f\ x)\ (f'\ x')\ ,$$
where
$$h_2\colon Link_{Pre}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ x\ x'\ .$$
There are three parts. Each part consists of a reduction followed by a derivation.

(a)
$$\begin{aligned}&Com\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ f\ f'\\ &\twoheadrightarrow i\ X\ Y\ X'\ Y'\ r\ s\ f\ f'\\ &\twoheadrightarrow \forall x\colon X\,.\,f'\ (r\ x) = s\ (f\ x)\ .\end{aligned}$$

$\therefore$

$$\dfrac{\dfrac{\Delta_3 \vdash Com\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ f\ f'_{(Ass)}}{\Delta_3 \vdash \forall x\colon X\,.\,f'\ (r\ x) = s\ (f\ x)}\ (=_{red})}{\dfrac{[X,\,x\,] \vdash x_{(Ass)}}{\Delta_3, [x] \vdash f'\ (r\ x) = s\ (f\ x)}\ (\forall E)}\ .$$

(b)

$$\begin{aligned} & Link_{Pre}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ x\ x' \\ & \twoheadrightarrow i\ X\ Y\ X'\ Y'\ r\ s\ x\ x' \\ & \twoheadrightarrow (\lambda x\,.\,\lambda x'\,.\,x' = r\ x)\ x\ x' \\ & \twoheadrightarrow x' = r\ x\ . \end{aligned}$$

$\therefore$

$$\dfrac{\Delta_4, [x,\,x',\,h_2\,] \vdash Link_{Pre}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ x\ x'_{(Ass)}}{\Delta_4, [x,\,x',\,h_2\,] \vdash x' = r\ x}\ (=_{red})\ .$$

(c)

$$\begin{aligned} & Link_{Post}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ (f\ x)\ (f'\ x') \\ & \twoheadrightarrow i\ X\ Y\ X'\ Y'\ r\ s\ (f\ x)\ (f'\ x') \\ & \twoheadrightarrow (\lambda y\,.\,\lambda y'\,.\,y' = s\ y)\ (f\ x)\ (f'\ x') \\ & \twoheadrightarrow f'\ x' = s\ (f\ x)\ . \end{aligned}$$

Use parts (a) and (b).

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta_3, [x] \vdash f'\ (r\ x) = s\ (f\ x) \quad \Delta_4, [x,\,x',\,h_2\,] \vdash x' = r\ x}{\Delta_3, [x,\,x',\,h_2\,] \vdash f'\ x' = s\ (f\ x)}\ (IE)}{\Delta_3, [x,\,x',\,h_2\,] \vdash Link_{Post}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ (f\ x)\ (f'\ x')}\ (=_{abs})}{\vdash \forall_{\Delta_3, [x,\,x',\,h_2\,]}\,.\,Link_{Post}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ (f\ x)\ (f'\ x')}\ (\forall I)_*\ .$$

3. Prove

$$\vdash \forall_{\Delta_5, [x,\,x',\,h_2\,]}\,.\,Link_{Post}\ X\ Y\ X'\ Y'\ (@^2_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ (f\ x)\ (f'\ x')\ ,$$

where

$$h_2\colon Link_{Pre}\ X\ Y\ X'\ Y'\ (@^2_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ x\ x'\ .$$

The proof is similar to that of step 2, except that it relates to $@^2_{Link}$ instead of $@^1_{Link}$, Δ_4 and Δ_5 instead of Δ_2 and Δ_3, and list membership instead of equality.

4. Prove

$$\Delta_2, [x,\, x',\, h_2 \colon Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x'\,] \vdash Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; (f'\; x')\;.$$

Use steps 2 and 3.

$$\dfrac{\dfrac{\begin{array}{c}\vdash \forall_{[X,Y,X',Y',r,s\,]}\,.\,\forall_{[f,f',c,x,x',h_2\,]}\,.\,Link_{Post}\; X\; Y\; X'\; Y'\; (@^1_{Link}\;\ldots)\; (f\; x)\; (f'\; x') \\ \vdash \forall_{[X,Y,X',Y',r,s\,]}\,.\,\forall_{[f,f',c,x,x',h_2\,]}\,.\,Link_{Post}\; X\; Y\; X'\; Y'\; (@^2_{Link}\;\ldots)\; (f\; x)\; (f'\; x')\end{array}}{[X,\, Y,\, X',\, Y',\, l\,] \vdash \forall_{[f,f',c,x,x',h_2\,]}\,.\,Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; (f'\; x')}\;(Link\,E_1)}{\Delta_2, [x,\, x',\, h_2\,] \vdash Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; (f'\; x')}\;(\forall I)_*\;.$$

5. Prove

$$\begin{array}{l}\Delta \vdash \forall x \colon X\,.\,\forall x' \colon X'\,.\,Pre'\; x'\; \wedge Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x' \rightarrow \\ \qquad \exists y' \colon Y'\,.\,Post'\; x'\; y'\; \wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; y'\;.\end{array}$$

Use steps 1 and 4.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\Delta_1, [x',\, h_1\,] \vdash Post'\; x'\; (f'\; x') \\ \Delta_2, [x,\, x',\, h_2\,] \vdash Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; (f'\; x')\end{array}}{\begin{array}{c}\Delta, [x,\, x',\, h' \colon Pre'\; x'\; \wedge Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x'\,] \vdash \\ Post'\; x'\; (f'\; x')\; \wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; (f'\; x')\end{array}}\;(\wedge I)}{\Delta, [x,\, x',\, h' \colon Post'\; x'\; y'\; \wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; y'\,]}\;(\exists I)}{\begin{array}{l}\Delta \vdash \forall x \colon X\,.\,\forall x' \colon X'\,.\,Pre'\; x'\; \wedge Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x' \rightarrow \\ \qquad \exists y' \colon Y'\,.\,Post'\; x'\; y'\; \wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; y'\;.\end{array}}\;(\rightarrow I)\;(\forall I)_*\;.$$

6. Reduce

$$\forall x \colon X\,.\,P_{15}\; X\; Y\; f\; X'\; Y'\; f'\; Pre'\; Post'\; s'\; l\; c\; x\; (f\; x)\;.$$

Now,

$$\begin{array}{l} P_{15}\; X\; Y\; f\; X'\; Y'\; f'\; Pre'\; Post'\; s'\; l\; c \\ \twoheadrightarrow P_{14}\; X\; Y\; f\; X'\; Y'\; f'\; Pre'\; Post'\; s'\; \underbrace{(@_{\surd Link}\; X\; Y\; f\; X'\; Y'\; f'\; l\; c)}_{\surd l} \\ \twoheadrightarrow P_{13}\; X\; Y\; f\; X'\; Y'\; f'\; \underbrace{(@_{Data}\; X'\; Y'\; Pre'\; Post')}_{d'}\; s'\; \surd l \\ \twoheadrightarrow P_{12}\; X\; Y\; f\; X'\; Y'\; f'\; \underbrace{(@_{\surd Data}\; X'\; Y'\; f'\; d'\; s')}_{\surd d'}\; \surd l \\ \twoheadrightarrow P_{11}\; X\; Y\; f\; X'\; Y'\; f'\; \underbrace{(@_{\surd DataLink}\; X\; Y\; f\; X'\; Y'\; f'\; \surd d'\; \surd l\,)}_{\surd dl} \\ \twoheadrightarrow \widehat{P}\; X\; Y\; f\; \underbrace{(@^1_{\surd Poset}\; X\; Y\; f\; X'\; Y'\; f'\; \surd dl\,)}_{\surd p} \end{array}$$

$$
\begin{aligned}
&\twoheadrightarrow Log_{Poset}\; X\; Y\; (Extract_{Poset}\; X\; Y\; f\; \surd p\,)\\
&\twoheadrightarrow Log_{Poset}\; X\; Y\; (@^{1}_{Poset}\; X\; Y\; X'\; Y'\; (Extract_{DataLink}\; X\; Y\; f\; X'\; Y'\; f'\; \surd dl\,))\\
&\twoheadrightarrow Log_{Poset}\; X\; Y\; (@^{1}_{Poset}\; X\; Y\; X'\; Y'\; (@_{DataLink}\; X\; Y\; X'\; Y'\; d'\; l))\\
&\twoheadrightarrow Log_{DataLink}\; X\; Y\; X'\; Y'\; (@_{DataLink}\; X\; Y\; X'\; Y'\; d'\; l)\\
&\twoheadrightarrow \lambda x\,.\,\lambda y\,.\,\forall x' \colon X'\,.\,(Data_{Pre}\; X'\; Y'\; d')\; x' \;\wedge Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x' \rightarrow\\
&\qquad \exists y' \colon Y'\,.\,(Data_{Post}\; X\; Y\; d')\; x'\; y' \;\wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; y\; y'\\
&\twoheadrightarrow \lambda x\,.\,\lambda y\,.\,\forall x' \colon X'\,.\,Pre'\; x' \;\wedge Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x' \rightarrow\\
&\qquad \exists y' \colon Y'\,.\,Post'\; x'\; y' \;\wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; y\; y'\;.
\end{aligned}
$$

$\therefore$

$$
\begin{aligned}
&\forall x \colon X\,.\,P_{15}\; X\; Y\; f\; X'\; Y'\; f'\; Pre'\; Post'\; s'\; l\; c\; x\; (f\; x)\\
&\twoheadrightarrow \forall x \colon X\,.\,\forall x' \colon X'\,.\,Pre'\; x' \;\wedge Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x' \rightarrow\\
&\qquad \exists y' \colon Y'\,.\,Post'\; x'\; y' \;\wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; y'\;.
\end{aligned}
$$

7. Finally, prove (6.6). Use step 6.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\Delta \vdash \forall x \colon X\,.\,\forall x' \colon X'\,.\,Pre'\; x' \;\wedge Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x' \rightarrow\\ \exists y' \colon Y'\,.\,Post'\; x'\; y' \;\wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; y'\end{array}}{\Delta \vdash \forall x\,.\,P_{15}\; X\; Y\; f\; X'\; Y'\; f'\; Pre'\; Post'\; s'\; l\; c\; x\; (f\; x)}\;(=_{abs})}{\vdash \forall_{[X,Y,f,X',Y',f',l,c]}\,.\,\forall_{[Pre',Post',s']}\,.\,\forall x\,.\,P_{15}\; X\; Y\; f\; X'\; Y'\; f'\; Pre'\; Post'\; s'\; l\; c\; x\; (f\; x)}\;(\forall I)_*}{[X,\, Y,\, f,\, X',\, Y',\, f',\, \surd l\,] \vdash \forall_{[Pre',Post',s']}\,.\,\forall x\,.\,P_{14}\; X\; Y\; f\; X'\; Y'\; f'\; Pre'\; Post'\; s'\; \surd l\;\; x\; (f\; x)}\;(\surd Link\;\; E_1)}{\vdash \forall_{[X',Y',Pre',Post']}\,.\,\forall_{[X,Y,f,\surd l\,,f',s']}\,.\,\forall x\,.\,P_{14}\; X\; Y\; f\; X'\; Y'\; f'\; Pre'\; Post'\; s'\; \surd l\;\; x\; (f\; x)}\;(\forall I)_*}{[X',\, Y',\, d'\,] \vdash \forall_{[X,Y,f,\surd l\,,f',s']}\,.\,\forall x\,.\,P_{13}\; X\; Y\; f\; X'\; Y'\; f'\; d'\; s'\; \surd l\;\; x\; (f\; x)}\;(Data\, E_1)}{\vdash \forall_{[X',Y',f',d',s']}\,.\,\forall_{[X,Y,f,\surd l\,]}\,.\,\forall x\,.\,P_{13}\; X\; Y\; f\; X'\; Y'\; f'\; d'\; s'\; \surd l\;\; x\; (f\; x)}\;(\forall I)_*}{[X',\, Y',\, f',\, \surd d'\,] \vdash \forall_{[X,Y,f,\surd l\,]}\,.\,\forall x\,.\,P_{12}\; X\; Y\; f\; X'\; Y'\; f'\; \surd d'\;\; \surd l\;\; x\; (f\; x)}\;(\surd Data\;\; E_1)}{\vdash \forall_{[X,Y,f,X',Y',f',\surd d'\,,\surd l\,]}\,.\,\forall x\,.\,P_{12}\; X\; Y\; f\; X'\; Y'\; f'\; \surd d'\;\; \surd l\;\; x\; (f\; x)}\;(\forall I)_*}{[X,\, Y,\, f,\, X',\, Y',\, f',\, \surd dl\,] \vdash \forall x\,.\,P_{11}\; X\; Y\; f\; X'\; Y'\; f'\; \surd dl\;\; x\; (f\; x)}\;(\surd DataLink\;\; E_1)}{\vdash \forall_{[X,Y,f,X',Y',f',\surd dl\,]}\,.\,\forall x\,.\,P_{11}\; X\; Y\; f\; X'\; Y'\; f'\; \surd dl\;\; x\; (f\; x)}\;(\forall I)_*\;.
$$

□

Coq Listing 74. *(Lemma 7)*

```
Lemma Lemma_Poset_1 :

forall X : Set,
forall Y : Set,
forall f : X -> Y,
forall X' : Set,
forall Y' : Set,
```

```
forall f' : X' -> Y',
forall cdl : CertDataLink X Y f X' Y' f',
forall x : X,

LogPoset X Y
  (ExtractPoset X Y f
     (Build_CertPoset_1 X Y f X' Y' f' cdl)) x (f x).

Proof.
   intros X Y f X' Y' f' cdl x.
   destruct cdl as [X Y f X' Y' f' cd' cl].
   destruct cd' as [X' Y' f' d' skol'].
   destruct d' as [X' Y' Pre' Post'].
   destruct cl as [X Y f X' Y' f' l com].
   simpl.
   unfold LogDataLink.LogDataLink.
   simpl.
   intros x' H.
   destruct H as [H1 H2].
   exists (f' x').
   split.
   exact (skol' x' H1).
   destruct l as [X Y X' Y' r s | X Y X' Y' r s].
   rewrite H2.
   exact (com x).
   exact (com x x' H2).
Qed.
```

Lemma 8. The logical interpretation of a data link poset, as extracted from a certified data link poset of the form

$$@^2_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ \surd p'\ ,$$

is provable on all objects x of X and $(f\,x)$ of Y, for all certified data link components $\surd dl$, all certified data link posets $\surd p'$, and all hypotheses that the logical interpretation of the data link poset extracted from $\surd p'$ is provable on all objects x' of X' and $(f'\,x')$ of Y', i.e.

$$\begin{aligned}\vdash \forall_{[X,Y,f,X',Y',f',\surd dl\,,\surd p'\,]}\,.\,(\forall x'\,.\,\widehat{P}\ X'\ Y'\ f'\ \surd p'\ x'\ (f'\ x')) \rightarrow \\ \forall x\,.\,P_{21}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ \surd p'\ x\ (f\ x)\ .\end{aligned} \tag{6.7}$$

Proof. In many respects, the proof is similar to that of Lemma 7, except that it relates to $@^2_{Poset}$ instead of $@^1_{Poset}$. However, this lemma has something that the other lemma does not have, and that is a recursive specification. After due consideration, the author decided against simply annotating the differences between the two proofs, choosing instead to virtually present the proof in full. As with Lemma 7, let

$$\Delta\ =_{df}\ \Delta_1 \cup \Delta_2\ ,$$

where

$$\Delta_1 =_{df} [X', Y', f', Pre', Post', s' : Skol\ X'\ Y'\ (@_{Data}\ X'\ Y'\ Pre'\ Post')\ f'\,]$$
$$\Delta_2 =_{df} [X, Y, f, X', Y', f', l, c : Com\ X\ Y\ X'\ Y'\ l\ f\ f'\,]\ .$$

1. Prove

$$\vdash \forall_{[X,Y,X',Y',r,s\,]} \cdot \forall_{[f,f',c,\surd p',ih',x,x',h_2\,]} \cdot Link_{Post}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ (f\ x)\ (f'\ x') \wedge \widehat{P}\ X'\ Y'\ f'\ \surd p'\ x'\ (f'\ x')\ ,$$

where

$$c : Com\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ f\ f'$$

and

$$h_2 : Link_{Pre}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ x\ x'\ .$$

(a) Establish $\widehat{P} X'\ Y'\ f'\ \surd p'\ x'\ (f'\ x')$.

$$\frac{[X', Y', f', \surd p', ih', x'\,] \vdash \forall x' : X' \,.\, \widehat{P}\ X'\ Y'\ f'\ \surd p'\ x'\ (f'\ x')_{(Ass)} \quad [X', x'\,] \vdash x'_{(Ass)}}{[X', Y', f', \surd p', ih', x'\,] \vdash \widehat{P}\ X'\ Y'\ f'\ \surd p'\ x'\ (f'\ x')}\ (\forall E)\ .$$

(b) Use step 2, part(c) of Lemma 7.

$$\frac{\dfrac{\vdash \forall_{\Delta_3,[x,x',h_2\,]} \,.\, Link_{Post}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ (f\ x)\ (f'\ x') \quad [X', Y', f', \surd p', ih', x'\,] \vdash \widehat{P}\ X'\ Y'\ f'\ \surd p'\ x'\ (f'\ x')}{[X, Y, X', Y', r, s\,], [f, f', c, \surd p', ih', x, x', h_2\,] \vdash Link_{Post}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ (f\ x)\ (f'\ x') \wedge \widehat{P}\ X'\ Y'\ f'\ \surd p'\ x'\ (f'\ x')}\ (\wedge I)}{\vdash \forall_{[X,Y,X',Y',r,s\,]} \cdot \forall_{[f,f',c,\surd p',ih',x,x',h_2\,]} \cdot Link_{Post}\ X\ Y\ X'\ Y'\ (@^1_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ (f\ x)\ (f'\ x') \wedge \widehat{P}\ X'\ Y'\ f'\ \surd p'\ x'\ (f'\ x')\ ,}\ (\forall I)_*$$

2. Prove

$$\vdash \forall_{[X,Y,X',Y',r,s\,]} \cdot \forall_{[f,f',c,\surd p',ih',x,x',h_2\,]} \cdot Link_{Post}\ X\ Y\ X'\ Y'\ (@^2_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ (f\ x)\ (f'\ x') \wedge \widehat{P}\ X'\ Y'\ f'\ \surd p'\ x'\ (f'\ x')\ ,$$

where

$$c : Com\ X\ Y\ X'\ Y'\ (@^2_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ f\ f'$$

and

$$h_2 : Link_{Pre}\ X\ Y\ X'\ Y'\ (@^2_{Link}\ X\ Y\ X'\ Y'\ r\ s)\ x\ x'\ .$$

The proof is similar to that of step 1, except that it relates to $@^2_{Link}$ instead of $@^1_{Link}$, and list membership instead of equality.

3. Prove

$$\Delta_2, [\surd p'\,,\, ih',\, x,\, x',\, h_2 \colon Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x'\,] \vdash \\ Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; (f'\; x')\; \wedge \widehat{P}\; X'\; Y'\; f'\; \surd p'\; x'\; (f'\; x')\;.$$

Use steps 1 and 2.

$$\dfrac{\begin{array}{l}\vdash \forall_{[X,Y,X',Y',r,s\,]} \cdot \forall_{[f,f',c,\surd p',ih',x,x',h_2\,]} \cdot \\ \quad Link_{Post}\; X\; Y\; X'\; Y'\; (@^1_{Link}\; \ldots)\; (f\; x)\; (f'\; x')\; \wedge \widehat{P}\; X'\; Y'\; f'\; \surd p'\; x'\; (f'\; x') \\ \vdash \forall_{[X,Y,X',Y',r,s\,]} \cdot \forall_{[f,f',c,\surd p',ih',x,x',h_2\,]} \cdot \\ \quad Link_{Post}\; X\; Y\; X'\; Y'\; (@^2_{Link}\; \ldots)\; (f\; x)\; (f'\; x')\; \wedge \widehat{P}\; X'\; Y'\; f'\; \surd p'\; x'\; (f'\; x')\end{array}}{\dfrac{\begin{array}{l}[X,\, Y,\, X',\, Y',\, l\,] \vdash \forall_{[f,f',c,\surd p',ih',x,x',h_2\,]} \cdot \\ \quad Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; (f'\; x')\; \wedge \widehat{P}\; X'\; Y'\; f'\; \surd p'\; x'\; (f'\; x')\end{array}}{\begin{array}{l}\Delta_2, [\surd p'\,,\, ih',\, x,\, x',\, h_2\,] \vdash \\ \quad Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; (f'\; x')\; \wedge \widehat{P}\; X'\; Y'\; f'\; \surd p'\; x'\; (f'\; x')\end{array}}\,(\forall I)_*\;.}\,(Link\, E_1)$$

4. Prove

$$\Delta, [\surd p'\,,\, ih'\,] \vdash \forall x \colon X\,.\, \forall x' \colon X'\,.\, Pre'\; x'\; \wedge Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x' \rightarrow \\ \exists y' \colon Y'\,.\, Post'\; x'\; y'\; \wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; y'\; \wedge \widehat{P}\; X'\; Y'\; f'\; \surd p'\; x'\; y'\;.$$

Use step 1 of Lemma 7 and step 3.

$$\dfrac{\dfrac{\dfrac{\begin{array}{l}\Delta_1, [x',\, h_1\,] \vdash Post'\; x'\; (f'\; x') \\ \Delta_2, [\surd p'\,,\, ih',\, x,\, x',\, h_2\,] \vdash Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; (f'\; x') \wedge \\ \quad \widehat{P}\; X'\; Y'\; f'\; \surd p'\; x'\; (f'\; x')\end{array}}{\begin{array}{l}\Delta, [\surd p'\,,\, ih',\, x,\, x',\, h' \colon Pre'\; x'\; \wedge Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x'\,] \vdash \\ \quad Post'\; x'\; (f'\; x')\; \wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; (f'\; x') \wedge \\ \quad\quad \widehat{P}\; X'\; Y'\; f'\; \surd p'\; x'\; (f'\; x')\end{array}}\,(\wedge I)}{\begin{array}{l}\Delta, [\surd p'\,,\, ih',\, x,\, x',\, h'\,] \vdash \exists y' \colon Y'\,.\, Post'\; x'\; y'\; \wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; y' \wedge \\ \quad \widehat{P}\; X'\; Y'\; f'\; \surd p'\; x'\; y'\end{array}}\,(\exists I)}{\begin{array}{l}\Delta, [\surd p'\,,\, ih'\,] \vdash \forall x \colon X\,.\, \forall x' \colon X'\,.\, Pre'\; x'\; \wedge Link_{Pre}\; X\; Y\; X'\; Y'\; l\; x\; x' \rightarrow \\ \quad \exists y' \colon Y'\,.\, Post'\; x'\; y'\; \wedge Link_{Post}\; X\; Y\; X'\; Y'\; l\; (f\; x)\; y' \wedge \\ \quad\quad \widehat{P}\; X'\; Y'\; f'\; \surd p'\; x'\; y'\end{array}}\,(\rightarrow I)\,(\forall I)_*\;.$$

5. Convert

$$\forall x\,.\, P_{25}\; X\; Y\; f\; X'\; Y'\; f'\; Pre'\; Post'\; s'\; l\; c\; \surd p'\; x\; (f\; x)\;.$$

Now,

$$P_{25}\; X\; Y\; f\; X'\; Y'\; f'\; Pre'\; Post'\; s'\; l\; c\; \surd p' \\ \twoheadrightarrow P_{24}\; X\; Y\; f\; X'\; Y'\; f'\; Pre'\; Post'\; s'\; \underbrace{(@_{\surd Link}\; X\; Y\; f\; X'\; Y'\; f'\; l\; c)}_{\surd l}\; \surd p'$$

$$
\begin{aligned}
&\twoheadrightarrow P_{23}\ X\ Y\ f\ X'\ Y'\ f'\ \underbrace{(@_{Data}\ X'\ Y'\ Pre'\ Post')}_{d'}\ s'\ \surd l\ \surd p' \\
&\twoheadrightarrow P_{22}\ X\ Y\ f\ X'\ Y'\ f'\ \underbrace{(@_{\surd Data}\ X'\ Y'\ f'\ d'\ s')}_{\surd d'}\ \surd l\ \surd p' \\
&\twoheadrightarrow P_{21}\ X\ Y\ f\ X'\ Y'\ f'\ \underbrace{(@_{\surd DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd d'\ \surd l)}_{\surd dl}\ \surd p' \\
&\twoheadrightarrow \widehat{P}\ X\ Y\ f\ \underbrace{(@^2_{\surd Poset}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl\ \surd p')}_{\surd p} \\
&\twoheadrightarrow Log_{Poset}\ X\ Y\ (Extract_{Poset}\ X\ Y\ f\ \surd p) \\
&\twoheadrightarrow Log_{Poset}\ X\ Y\ (@^2_{Poset}\ X\ Y\ X'\ Y' \\
&\qquad (Extract_{DataLink}\ X\ Y\ f\ X'\ Y'\ f'\ \surd dl)\ (Extract_{Poset}\ X'\ Y'\ f'\ \surd p')) \\
&\twoheadrightarrow Log_{Poset}\ X\ Y\ (@^2_{Poset}\ X\ Y\ X'\ Y' \\
&\qquad (@_{DataLink}\ X\ Y\ X'\ Y'\ d'\ l)\ (Extract_{Poset}\ X'\ Y'\ f'\ \surd p')) \\
&\twoheadrightarrow \lambda x\,.\,\lambda y\,.\,\forall x'\colon X'\,.\,(Data_{Pre}\ X'\ Y'\ d')\ x'\ \wedge Link_{Pre}\ X\ Y\ X'\ Y'\ l\ x\ x' \rightarrow \\
&\qquad \exists y'\colon Y'\,.\,(Data_{Post}\ X\ Y\ d')\ x'\ y'\ \wedge Link_{Post}\ X\ Y\ X'\ Y'\ l\ y\ y' \wedge \\
&\qquad\quad Log_{Poset}\ X'\ Y'\ (Extract_{Poset}\ X'\ Y'\ f'\ \surd p')\ x'\ y' \\
&\twoheadleftarrow\!\!\twoheadrightarrow \lambda x\,.\,\lambda y\,.\,\forall x'\colon X'\,.\,Pre'\ x'\ \wedge Link_{Pre}\ X\ Y\ X'\ Y'\ l\ x\ x' \rightarrow \\
&\qquad \exists y'\colon Y'\,.\,Post'\ x'\ y'\ \wedge Link_{Post}\ X\ Y\ X'\ Y'\ l\ y\ y'\ \wedge \widehat{P}\ X'\ Y'\ f'\ \surd p'\ x'\ y'\ .
\end{aligned}
$$

$\therefore$

$$
\begin{aligned}
&\forall x\,.\,P_{25}\ X\ Y\ f\ X'\ Y'\ f'\ Pre'\ Post'\ s'\ l\ c\ \surd p'\ x\ (f\ x) \\
&\twoheadleftarrow\!\!\twoheadrightarrow \forall x\colon X\,.\,\forall x'\colon X'\,.\,Pre'\ x'\ \wedge Link_{Pre}\ X\ Y\ X'\ Y'\ l\ x\ x' \rightarrow \\
&\qquad \exists y'\colon Y'\,.\,Post'\ x'\ y'\ \wedge Link_{Post}\ X\ Y\ X'\ Y'\ l\ (f\ x)\ y'\ \wedge \widehat{P}\ X'\ Y'\ f'\ \surd p'\ x'\ y'\ .
\end{aligned}
$$

6. Finally, prove (6.7). Use step 5.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta, [\sqrt{p'}\,,\, ih'] \vdash \forall x\colon X\,.\,\forall x'\colon X'\,.\,Pre'\ x'\ \wedge Link_{Pre}\ X\ Y\ X'\ Y'\ l\ x\ x' \rightarrow \exists y'\colon Y'\,.\,Post'\ x'\ y'\ \wedge Link_{Post}\ X\ Y\ X'\ Y'\ l\ (f\ x)\ y'\ \wedge \widehat{P}\ X'\ Y'\ f'\ \sqrt{p'}\ x'\ y'}{\Delta, [\sqrt{p'}\,,\, ih'] \vdash \forall x\,.\,P_{25}\ X\ Y\ f\ X'\ Y'\ f'\ Pre'\ Post'\ s'\ l\ c\ \sqrt{p'}\ x\ (f\ x)}\ (=_{conv})}{\vdash \forall_{[X,Y,f,X',Y',f',l,c]}\,.\,\forall_{[Pre',Post',s',\sqrt{p'}\,,ih']}\,.\,\forall x\,.\,P_{25}\ X\ Y\ f\ X'\ Y'\ f'\ Pre'\ Post'\ s'\ l\ c\ \sqrt{p'}\ x\ (f\ x)}\ (\forall I)_*}{[X,\ Y,\ f,\ X',\ Y',\ f',\ \sqrt{l}\ ] \vdash \forall_{[Pre',Post',s',\sqrt{p'}\,,ih']}\,.\,\forall x\,.\,P_{24}\ X\ Y\ f\ X'\ Y'\ f'\ Pre'\ Post'\ s'\ \sqrt{l}\ \sqrt{p'}\ x\ (f\ x)}\ (\sqrt{Link}\ E_1)}{\vdash \forall_{[X',Y',Pre',Post']}\,.\,\forall_{[X,Y,f,\sqrt{l}\,,f',s',\sqrt{p'}\,,ih']}\,.\,\forall x\,.\,P_{24}\ X\ Y\ f\ X'\ Y'\ f'\ Pre'\ Post'\ s'\ \sqrt{l}\ \sqrt{p'}\ x\ (f\ x)}\ (\forall I)_*}{[X',\ Y',\ d'\ ] \vdash \forall_{[X,Y,f,\sqrt{l}\,,f',s',\sqrt{p'}\,,ih']}\,.\,\forall x\,.\,P_{23}\ X\ Y\ f\ X'\ Y'\ f'\ d'\ s'\ \sqrt{l}\ \sqrt{p'}\ x\ (f\ x)}\ (Data\ E_1)}{\vdash \forall_{[X',Y',f',d',s']}\,.\,\forall_{[X,Y,f,\sqrt{l}\,,\sqrt{p'}\,,ih']}\,.\,\forall x\,.\,P_{23}\ X\ Y\ f\ X'\ Y'\ f'\ d'\ s'\ \sqrt{l}\ \sqrt{p'}\ x\ (f\ x)}\ (\forall I)_*}{[X',\ Y',\ f',\ \sqrt{d'}\ ] \vdash \forall_{[X,Y,f,\sqrt{l}\,,\sqrt{p'}\,,ih']}\,.\,\forall x\,.\,P_{22}\ X\ Y\ f\ X'\ Y'\ f'\ \sqrt{d'}\ \sqrt{l}\ \sqrt{p'}\ x\ (f\ x)}\ (\sqrt{Data}\ E_1)}{\vdash \forall_{[X,Y,f,X',Y',f',\sqrt{d'}\,,\sqrt{l}\,]}\,.\,\forall_{[\sqrt{p'}\,,ih']}\,.\,\forall x\,.\,P_{22}\ X\ Y\ f\ X'\ Y'\ f'\ \sqrt{d'}\ \sqrt{l}\ \sqrt{p'}\ x\ (f\ x)}\ (\forall I)_*}{[X,\ Y,\ f,\ X',\ Y',\ f',\ \sqrt{dl}\ ] \vdash \forall_{[\sqrt{p'}\,,ih']}\,.\,\forall x\,.\,P_{21}\ X\ Y\ f\ X'\ Y'\ f'\ \sqrt{dl}\ \sqrt{p'}\ x\ (f\ x)}\ (\sqrt{D'Link}\ E_1)}{\vdash \forall_{[X,Y,f,X',Y',f',\sqrt{dl}\,,\sqrt{p'}\,]}\,.\,(\forall x'\,.\,\widehat{P}\ X'\ Y'\ f'\ \sqrt{p'}\ x'\ (f'\ x')) \rightarrow \forall x\,.\,P_{21}\ X\ Y\ f\ X'\ Y'\ f'\ \sqrt{dl}\ \sqrt{p'}\ x\ (f\ x)}\ (\forall I)_*\,.$$

For convenience, the inductive hypothesis

$$\forall x'\,.\,\widehat{P}\ X'\ Y'\ f'\ \sqrt{p'}\ x'\ (f'\ x')$$

is bound to variable ih'.

□

Coq Listing 75. *(Lemma 8)*

```
Lemma Lemma_Poset_2 :

forall X : Set,
forall Y : Set,
forall f : X -> Y,
forall X' : Set,
forall Y' : Set,
forall f' : X' -> Y',
forall cdl : CertDataLink X Y f X' Y' f',
forall cp' : CertPoset X' Y' f',
(forall x' : X', LogPoset X' Y' (ExtractPoset X' Y' f' cp') x' (f' x')) ->
forall x : X,

LogPoset X Y
   (ExtractPoset X Y f
      (Build_CertPoset_2 X Y f X' Y' f' cdl cp')) x (f x).

Proof.
   intros X Y f X' Y' f' cdl cp' IH' x.
```

```
  destruct cdl as [X Y f X' Y' f' cd' cl].
  destruct cd' as [X' Y' f' d' skol'].
  destruct d' as [X' Y' Pre' Post'].
  destruct cl as [X Y f X' Y' f' l com].
  simpl.
  intros x' H.
  destruct H as [H1 H2].
  exists (f' x').
  split.
  exact (skol' x' H1).
  destruct l as [X Y X' Y' r s | X Y X' Y' r s].
  split.
  rewrite H2.
  exact (com x).
  exact (IH' x').
  split.
  exact (com x x' H2).
  exact (IH' x').
Qed.
```

□

Lemma 9. The logical interpretation of a data link poset, as extracted from a certified data link poset of the form

$$@^3_{\surd Poset}\ X\ Y\ f\ \surd p_1\ \surd p_2\ ,$$

is provable on all objects x of X and $(f\,x)$ of Y, for all certified data link posets $\surd p_1$ and $\surd p_2$, and all hypotheses that the logical interpretations of the data link posets, as extracted from $\surd p_1$ and $\surd p_2$, are provable on all objects x and $(f\,x)$, i.e.

$$\begin{aligned}\vdash \forall_{[X,Y,f,\surd p_1\,,\surd p_2\,]}\,.\,(\forall x\colon X\,.\,\widehat{P}\ X\ Y\ f\ \surd p_1\ x\ (f\,x)) \to \\ (\forall x\colon X\,.\,\widehat{P}\ X\ Y\ f\ \surd p_2\ x\ (f\,x)) \to \forall x\colon X\,.\,P_{31}\ X\ Y\ f\ \surd p_1\ \surd p_2\ x\ (f\,x)\ .\end{aligned} \tag{6.8}$$

Proof. There are three steps.

1. Prove

$$\begin{aligned}[X,\,Y,\,f,\,\surd p_1\,,\,\surd p_2\,,\,ih_1\colon \forall x\colon X\,.\,\widehat{P}\ X\ Y\ f\ \surd p_1\ x\ (f\,x),\,ih_2\colon \forall x\colon X\,.\,\ldots\,]\vdash \\ \forall x\colon X\,.\,\widehat{P}\ X\ Y\ f\ \surd p_1\ x\ (f\,x)\ \wedge \widehat{P}\ X\ Y\ f\ \surd p_2\ x\ (f\,x)\ .\end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{[X,\,Y,\,f,\,\surd p_1\,,\,ih_1\,]\vdash \forall x\colon X\,.\,\widehat{P}\ X\ Y\ f\ \surd p_1\ x\ (f\,x)_{(Ass)} \quad [X,\,x\,]\vdash x_{(Ass)}}{[X,\,Y,\,f,\,\surd p_1\,,\,ih_1,\,x\,]\vdash \widehat{P}\ X\ Y\ f\ \surd p_1\ x\ (f\,x)}\,(\forall E) \quad \dfrac{[X,\,Y,\,f,\,\surd p_2\,,\,ih_2\,]\vdash \forall x\colon X\,.\,\widehat{P}\ X\ Y\ f\ \surd p_2\ x\ (f\,x)_{(Ass)} \quad [X,\,x\,]\vdash x_{(Ass)}}{[X,\,Y,\,f,\,\surd p_2\,,\,ih_2,\,x\,]\vdash \widehat{P}\ X\ Y\ f\ \surd p_2\ x\ (f\,x)}\,(\forall E)}{[X,\,Y,\,f,\,\surd p_1\,,\,\surd p_2\,,\,ih_1,\,ih_2,\,x\,]\vdash \widehat{P}\ X\ Y\ f\ \surd p_1\ x\ (f\,x)\ \wedge \widehat{P}\ X\ Y\ f\ \surd p_2\ x\ (f\,x)}\,(\wedge I)}{[X,\,Y,\,f,\,\surd p_1\,,\,\surd p_2\,,\,ih_1,\,ih_2\,]\vdash \forall x\colon X\,.\,\widehat{P}\ X\ Y\ f\ \surd p_1\ x\ (f\,x)\ \wedge \widehat{P}\ X\ Y\ f\ \surd p_2\ x\ (f\,x)\ .}\,(\forall I)\ .$$

2. Convert

$$\forall x\colon X\,.\,P_{31}\;X\;Y\;f\;\surd p_1\;\surd p_2\;x\,(f\,x)\;.$$

Now,

$$\begin{aligned}
&P_{31}\;X\;Y\;f\;\surd p_1\;\surd p_2\\
&\twoheadrightarrow \widehat{P}\;X\;Y\;f\;(@^3_{\surd Poset}\;X\;Y\;f\;\surd p_1\;\surd p_2\,)\\
&\twoheadrightarrow Log_{Poset}\;X\;Y\;(Extract_{Poset}\;X\;Y\;f\;(@^3_{\surd Poset}\;X\;Y\;f\;\surd p_1\;\surd p_2\,))\\
&\twoheadrightarrow Log_{Poset}\;X\;Y\;(@^3_{\surd Poset}\;X\;Y\\
&\qquad (Extract_{Poset}\;X\;Y\;f\;\surd p_1\,)\;(Extract_{Poset}\;X\;Y\;f\;\surd p_2\,))\\
&\twoheadrightarrow \lambda x\,.\,\lambda y\,.\,Log_{Poset}\;X\;Y\;(Extract_{Poset}\;X\;Y\;f\;\surd p_1\,)\;x\;y\;\wedge\\
&\qquad Log_{Poset}\;X\;Y\;(Extract_{Poset}\;X\;Y\;f\;\surd p_2\,)\;x\;y\;.
\end{aligned}$$

$\therefore$

$$\begin{aligned}
&\forall x\colon X\,.\,P_{31}\;X\;Y\;f\;\surd p_1\;\surd p_2\;x\,(f\,x)\\
&\twoheadrightarrow \forall x\colon X\,.\,Log_{Poset}\;X\;Y\;(Extract_{Poset}\;X\;Y\;f\;\surd p_1\,)\;x\,(f\,x)\;\wedge\\
&\qquad Log_{Poset}\;X\;Y\;(Extract_{Poset}\;X\;Y\;f\;\surd p_2\,)\;x\,(f\,x)\\
&\twoheadleftarrow \forall x\colon X\,.\,\widehat{P}\;X\;Y\;f\;\surd p_1\;x\,(f\,x)\;\wedge\widehat{P}\;X\;Y\;f\;\surd p_2\;x\,(f\,x)\;.
\end{aligned}$$

3. Finally, prove (6.8). Use steps 1 and 2.

$$\dfrac{\dfrac{[X,\,Y,\,f,\,\surd p_1\,,\,\surd p_2\,,\,ih_1,\,ih_2\,]\vdash \forall x\colon X\,.\,\widehat{P}\;X\;Y\;f\;\surd p_1\;x\,(f\,x)\;\wedge\widehat{P}\;X\;Y\;f\;\surd p_2\;x\,(f\,x)}{[X,\,Y,\,f,\,\surd p_1\,,\,\surd p_2\,,\,ih_1,\,ih_2\,]\vdash \forall x\colon X\,.\,P_{31}\;X\;Y\;f\;\surd p_1\;\surd p_2\;x\,(f\,x)}\,(=_{conv})}{\begin{array}{l}\vdash \forall_{[X,\,Y,\,f,\,\surd p_1\,,\,\surd p_2\,]}\,.\,(\forall x\colon X\,.\,\widehat{P}\;X\;Y\;f\;\surd p_1\;x\,(f\,x))\rightarrow\\ \qquad(\forall x\colon X\,.\,\widehat{P}\;X\;Y\;f\;\surd p_2\;x\,(f\,x))\rightarrow\forall x\colon X\,.\,P_{31}\;X\;Y\;f\;\surd p_1\;\surd p_2\;x\,(f\,x)\end{array}}\,(\rightarrow I)_*(\forall I)_*\;.$$

□

Coq Listing 76. *(Lemma 9)*

```
Theorem Lemma_Poset_3 :

forall X : Set,
forall Y : Set,
forall f : X -> Y,
forall cp1 : CertPoset X Y f,
forall cp2 : CertPoset X Y f,
(forall x : X, LogPoset X Y (ExtractPoset X Y f cp1) x (f x)) ->
(forall x : X, LogPoset X Y (ExtractPoset X Y f cp2) x (f x)) ->
forall x : X,

LogPoset X Y
```

```
   (ExtractPoset X Y f
      (Build_CertPoset_3 X Y f cp1 cp2)) x (f x).

Proof.
   intros X Y f cp1 cp2 IH1 IH2 x.
   simpl.
   split.
   exact (IH1 x).
   exact (IH2 x).
Qed.
```

This completes the proof of the logical interpretation theorem.

6.4 Concrete Example

This section contains the second part of a two-part example about a transformation between the simple metamodels of UML and SQL, as described in Section 5.4. In the first part, a monolithic proof of the logical interpretation of $t_{ModelSchema}$, i.e.

$$Log_{Tran}\, Model\, Schema\, t_{ModelSchema}\ ,$$

was derived by hand, and a certified program that implements it was computed by Coq. In this section, an alternative proof is established, based on the theory of certified ordered model transformations. The two proofs are strikingly different, and so are the two certified programs. The first proof is based on type inference, whereas the second proof is based largely on construction.

Coq Listing 77. *(Certified Program)* The proof starts with five lemmas, one for each of the skolemised data components `Skol`, and link components `Com`. These components are similar to the fundamental building blocks of the monolithic proof. The lemmas are followed by the construction of every $\surd Data$, $\surd Link$, $\surd DataLink$, $\surd Poset$ and $\surd Tran$ component, up to and including $\surd t_{ModelSchema}$, the certified ordered model transformation. Finally, the theorem is asserted and proved in one step, by applying `Theorem_Tran`, the Coq encoding of the proof of the logical interpretation theorem.

```
(* Skol *)

Lemma skol_Model_Schema :
   Skol Model Schema d_Model_Schema f_Model.
Proof.
   intros m H.
   reflexivity.
Qed.

Lemma skol_Class_Table :
   Skol Class Table d_Class_Table f_Class.
Proof.
```

```
   intros m H.
   split.
   reflexivity.
   assumption.
Qed.

Lemma skol_Attribute_Column :
   Skol Attribute Column d_Attribute_Column f_Attribute.
Proof.
   intros m H.
   split.
   reflexivity.
   assumption.
Qed.

(* Com *)

Lemma com_Model_Schema :
   Com Model Schema Class Table l_Model_Schema f_Model f_Class.
Proof.
   simpl.
   intros m c H.
   induction (R_Classes m).
   elim H.
   destruct H.
   rewrite H.
   left.
   reflexivity.
   right.
   exact (IHl H).
Qed.

Lemma com_Class_Table :
   Com Class Table Attribute Column l_Class_Table f_Class f_Attribute.
Proof.
   simpl.
   intros c a H.
   induction (R_Attributes c).
   elim H.
   destruct H.
   rewrite H.
   left.
   reflexivity.
   right.
   exact (IHl H).
Qed.

(* Certified Data *)
```

```
Definition cd_Model_Schema : CertData Model Schema f_Model :=
   Build_CertData Model Schema f_Model d_Model_Schema skol_Model_Schema.

Definition cd_Class_Table : CertData Class Table f_Class :=
   Build_CertData Class Table f_Class d_Class_Table skol_Class_Table.

Definition cd_Attribute_Column : CertData Attribute Column f_Attribute :=
   Build_CertData Attribute Column f_Attribute d_Attribute_Column
      skol_Attribute_Column.

(* Certified Link *)

Definition cl_Model_Schema : CertLink Model Schema f_Model Class Table f_Class :=
   Build_CertLink Model Schema f_Model Class Table f_Class l_Model_Schema
      com_Model_Schema.

Definition cl_Class_Table :
   CertLink Class Table f_Class Attribute Column f_Attribute :=
   Build_CertLink Class Table f_Class Attribute Column f_Attribute l_Class_Table
      com_Class_Table.

(* Certified DataLink *)

Definition cdl_Model_Schema :
   CertDataLink Model Schema f_Model Class Table f_Class :=
   Build_CertDataLink Model Schema f_Model Class Table f_Class cd_Class_Table
      cl_Model_Schema.

Definition cdl_Class_Table :
   CertDataLink Class Table f_Class Attribute Column f_Attribute :=
   Build_CertDataLink Class Table f_Class Attribute Column f_Attribute
      cd_Attribute_Column cl_Class_Table.

(* Certifed Poset *)

Definition cp_Class_Table : CertPoset Class Table f_Class :=
   Build_CertPoset_1 Class Table f_Class Attribute Column f_Attribute
      cdl_Class_Table.

Definition cp_Model_Schema : CertPoset Model Schema f_Model :=
   Build_CertPoset_2 Model Schema f_Model Class Table f_Class cdl_Model_Schema
      cp_Class_Table.

(* Certified Tran *)

Definition ct_Model_Schema : CertTran Model Schema f_Model :=
   Build_CertTran_2 Model Schema f_Model cd_Model_Schema cp_Model_Schema.

(* Theorem *)
```

```
Theorem UmlSql : LogTran Model Schema t_Model_Schema.
Proof.
   apply (Theorem_Tran Model Schema f_Model ct_Model_Schema).
Qed.

> Print UmlSql.
UmlSql =
Theorem_Tran Model Schema f_Model ct_Model_Schema
     : LogTran Model Schema t_Model_Schema
```

7 Related Work

The rise of the model as a first class entity in software engineering, sparked an intense period of research into the design and development of model transformations in the 1990s and 2000s, so there is a lot of related work to consider. This chapter starts by discussing the foundational work on which much of what followed was based. It continues with a selection of related work in the fields of rewriting logic, type theory, graph theory and category theory, and concludes with a brief discussion of other miscellaneous related work. The body of work in this field is so large and diverse that the author initially found it difficult to categorise. Although there is a feature-based survey of approaches to model transformations by Czarnecki [45], on which the author considered basing the entire chapter, the relationship between features and research is so complex as to make any categorisation based on it untenable.

7.1 Foundational Work

Before the use of model driven engineering became widespread, a considerable amount of effort was expended in attempting to formalise the semantics of object-oriented programming languages. This work—which is summarised in the following sections—predates that of any research into the formal semantics of models, but is closely related to it.

7.1.1 Objects and Subtypes

Reynolds [118] compares and contrasts two approaches to data abstraction: user-defined types, as developed by Morris [92] and others, and procedural data structures, as developed by Reynolds [117] himself. In the first approach, an abstract kind of data is characterised by a type, which defines both the internal representation of the data, and the external means by which it is accessed, that is to say via a primitive set of operations. Within reason, the former can be changed without affecting the latter. In the second approach, an abstract kind of data is characterised by the set of primitive operations that can be performed on the data; and a particular item of data is defined by an internal representation of the data (which may vary from one item to another) and a collection of procedures for performing operations on that representation. In comparison to user-defined types, procedural data structures provide a decentralised form of data abstraction. However, the decentralisation

comes at a price, in that the procedures can only ever access the representation of one item of data.

Reynolds [119] also provides a categorical treatment of implicit type conversions for a simple imperative language. Consider, for example, the problem of assigning the sum of two integer variables m and n to a real variable x, in languages which do and do not support implicit type conversion from integer to real. In the latter language, two solutions are possible depending on whether conversions precede or follow summation, i.e.

$$x = real(m) + real(n)$$
$$x = real(m + n)\ .$$

In the former language, where programmers just write $x = m + n$, the language must unambiguously define which of the two assignment statements hold. Reynolds uses category theory to capture the relationships that exist between generic operations like + and implicit conversions like $real()$.

7.1.2 Type Inference

Rémy [116] describes a system for inferring the types of labelled products, i.e. records, in an extension of ML. In languages such as ML, which do not support records, data structures are built from product types, e.g.

$$(\text{“}John\text{”},\ \text{“}Smith\text{”},\ 21)\ ,$$

whereas in languages which do support records, the same structure can be written

$$\{forename = \text{“}John\text{”};\ surname = \text{“}Smith\text{”};\ age = 21\}\ .$$

Clearly, the record is more readable than the product. Rémy initially defines records as partial functions from labels L to types T, i.e. as elements of type $L \to T$, but later as total functions of type $L \to T \cup \{abs\}$, where abs is the type of an absent field. In Rémy's system, the type of a record with labels a, b and c, where a is the integer constant 1, b is absent, and c is the Boolean constant $true$, is given by

$$a \to integer,\ b \to abs,\ c \to boolean\ .$$

Rémy also describes a function $\_.a$ for reading the value of a field labelled a in any record where its value is defined, so that e.g.

$$\_.a\ \{a = 1;\ \ldots\} \to 1.$$

In applying $\_.a$, it is immaterial as to whether the values of fields other than a are defined.

Wand [138] extends Rémy's type inference system for records with finite label sets, to a system with potentially infinite label sets, and establishes a scheme for interpreting the language of classes (classes, objects, methods and single inheritance) in the language of records. In this scheme, a class is modelled as a function, which takes a set of variables

$$x_1, \ldots, x_n$$

as input, to a record

$$\{a_1 = M_1; \ldots; a_k = M_k\}$$

as output, where $a_1, \ldots, a_k$ are the names, and $M_1, \ldots, M_k$ are the bodies, of a set of publicly accessible methods for processing the variables $x_1, \ldots, x_n$ in some sense, while at the same time hiding them from public view. Further, inheritance is modelled by a function which takes a set of variables $x_1, \ldots, x_n$, and a variable *self* as input, to a record

$$(P(Q_1 \ldots Q_p)\ self)\ with\ \{a_1 = M_1; \ldots a_k = M_k\}$$

as output, i.e. the extension of the parent record $P(Q_1 \ldots Q_p)$ *self* with the child methods in $\{a_1 = M_1; \ldots a_k = M_k\}$, where P is the parent class, and $Q_1 \ldots Q_p$ are expressions which determine how P is instantiated. The value of *self*, as seen by the methods of P, is then defined to be the value of *self* for the entire record.

Ohori and Buneman [103] show how the fundamental concepts of object-oriented languages, i.e. data abstraction and method inheritance, can be supported in a static type system for an ML-like language, by defining a type inference system for a core language (with rules for record construction, field selection, and field modification); extending it to support class declarations; showing that the extended type system is sound; and developing an algorithm for statically determining the types of programs in the extended language.

7.1.3 Coherence

There was considerable interest in object oriented languages in the late 1980s, not only in the programming languages community, but also in the software engineering community. To many practitioners, an object oriented language is one that supports inheritance on subtypes, where a subtype is based on the intuition that if a function can be applied to an argument of type τ, then it can also be applied to an argument of type σ, where σ is a subtype of τ, i.e. $\sigma \leq \tau$; and that, in particular, if σ and τ are record types, where σ contains not only the same fields as τ, but also extra fields that τ does not have, then there is nothing conceptually difficult about allowing a function on τ to be applied on σ, since all the function has to do is to ignore the extra fields. Bruce and Longo [27] give a precise mathematical meaning to inheritance through the language *Bounded Fun* (Cardelli and Wergner [35]) using a generalisation of partial equivalence relations known as ω-sets.

Breazu-Tannen, Coquand, Gunter and Scedrov [25] discuss an approach to the semantics of inheritance based on the relation $\leq$ between types, which is defined in such a way that whenever the judgement $s \leq t$ is provable for types s and t, an expression of type s can also be considered to be an expression of type t. This is usually expressed by the *subsumption* rule: if an expression e is of type s, and $s \leq t$, then e is also of type t. The consequences of including the subsumption rule in a type system are profound, because it introduces an element of indeterminism which would otherwise not be there; programs can no longer be type-checked in just one way. Instead of interpreting a system with the subsumption rule directly, Breazu-Tannen *et al* translate it to the polymorphic lambda calculus by induction on the height of a derivation of the type of a term. The translation is shown to be coherent in the sense that no matter which derivation is used, the result is always the same up to provable equality in the target calculus.

Curien and Ghelli [43] address the problem of coherence in a system with subsumption, using a proof theoretical, rewriting approach based on the calculus $F_{\leq}$, a second order lambda calculus with bounded quantification derived from the language *Fun* (Cardelli and Wegner [35]). In a context where terms can be proved to be well-typed in many different ways, Curien and Ghelli identify and solve three seemingly unrelated problems: proving that different semantic interpretations of the same term are equivalent in some sense; proving that a most general type can be assigned to each well-typed term; and finding a deterministic type-checking algorithm which is sound and compete. The syntax and typing rules of two related systems are discussed: $F_{\leq}$ and $cF_{\leq}$, the latter having an additional sort—the sort of coercions—which is used to codify subtyping proofs in $F_{\leq}$. The basic idea is that there is a one-to-one correspondence between the proofs in $F_{\leq}$ and $cF_{\leq}$; and given that there is also a one-to-one correspondence between the terms and proofs in $cF_{\leq}$, it follows that the terms of $cF_{\leq}$ are isomorphic to the proofs of $F_{\leq}$. Curien and Ghelli investigate this isomorphism by means of a coerce reduction (or rewriting) system in $cF_{\leq}$, and develop a type-checking algorithm in $F_{\leq}$, which they prove to be sound and complete.

7.1.4 Record Calculi

Type systems for record structures received a lot of attention in the 1980s because they provided the foundations for typing expressions in object oriented languages. Cardelli [32] defined the basic notions of record types for a type system with fixed-sized records, and later, with Wegner [35], defined a system in which a program can polymorphically work over the subtypes B of a given record type A, preserving the fields that are in B but not A. However, in neither of these systems is it possible to manipulate records by adding or removing fields. Later still, Cardelli and Mitchell [34] describe a type system in which this is possible, through a collection of operations for creating and manipulating record structures. A record value is defined by a map from labels to values (where values may be of different types), and three basic operations with suitable constraints, i.e.

- extension $\langle r \mid x = a \rangle$, which adds a field labelled x and value a to a record r, provided x does not already exist;
- restriction $r \backslash x$, which removes a field labelled x from record r; and
- extraction $r.x$, which extracts the value of a field labelled x from record r, providing x exists.

Similarly, three basic operations are defined on record types (the types of record values), where a record type captures positive information about what fields the members of that record type must have, and negative information about what fields the members of that record type must not have. For example, $\langle\langle\ \rangle\rangle$ denotes the type of all records, $\langle\langle x\colon int \rangle\rangle$ denotes the type of all records which have at least a field labelled x, and $\langle\langle\ \rangle\rangle \backslash x$ denotes the type of all records which do not have a field labelled x. With these definitions in place, Cardelli and Mitchell define the terms of the type system, and the rules of record type formation and equivalence, record subtyping and type inference.

Rémy [115] asserts that a functional language that supports record extension also supports record concatenation (or merging) for free. He notes, however, that different semantics apply when records contain the same field: either the concatenation is rejected, as in the case of *symmetric* concatenation, or the value of the field is taken from the second record, as in the case of *asymmetric* concatenation. Rémy studies his assertion by defining an untyped λ-calculus L with record extension, i.e.

$$M =_{df} x \mid \lambda x \,.\, M \mid M\ M \mid \{\,\} \mid \{M \text{ with } a = M\} \mid M.a \ ,$$

and an untyped λ-calculus $L^{\|}$ with concatenation operator $\|$, i.e.

$$M =_{df} x \mid \lambda x \,.\, M \mid M\ M \mid \{\,\} \mid \{a = M\} \mid M \parallel M \mid M.a \ ,$$

where $L^{\|}$ is similar to L except that it eschews record extension for one-field records, which he considers more primitive in the face of concatenation, since

$$\{M \text{ with } a = N\} \equiv M \parallel \{a = N\} \ .$$

Rémy defines a translation between $L^{\|}$ and L which works for both symmetric and asymmetric concatenation, and then adapts it to meet the needs of a typed framework. Finally, he applies the theory to a record extension of ML.

Cardelli [33] shows how a calculus with extensible records can be translated into a calculus without extensible records, i.e. one that is devoid of record primitives altogether. He also shows that the translation is well-behaved in the sense that it preserves typing, subtyping and equality. Cardelli suggests that a simple calculus of subtyping should be used as the basis for studying the ever more divergent and complex calculi of extensible records. His working hypothesis is that "every reasonable calculus with extensible records should be reducible to a calculus without extensible records, via a well-behaved translation". His view is that a more fundamental framework is required to be able to compare and contrast different calculi.

The procedure for checking the subtype relation between $F_{\leq}$ types was thought to terminate on all inputs. However, a subtle bug in a supposed proof led Pierce [108] to show that $F_{\leq}$, the minimal bounded version of the language *Fun*, which combines bounded polymorphism with a calculus of subtyping, is undecidable. This came as a surprise to the many theorists who had studied, extended and applied the language since its inception in 1985. By encoding an $F_{\leq}$ subtyping statement as a two-counter Turing machine T, via an intermediate abstraction called a rowing machine $R(T)$, Pierce shows that T halts if and only if $R(T)$ halts, from which he deduces that $F_{\leq}$ subtyping, and as a consequence $F_{\leq}$ type checking, are both undecidable.

7.1.5 Object Calculi

Abadi and Cardelli [7] develop a theory of objects as a means of formalising the fundamental features of object oriented languages, in much the same way as various λ-calculi have been used to formalise the fundamental features of procedural languages. Their approach is different from other authors in that they model objects directly, rather than as adjuncts

of functions. Their book starts with a review of the standard concepts of class-based and object-based languages, [1] the design spaces of which Abadi and Cardelli see as at opposite ends of a continuum. They argue that "the achievement of object-based languages is to make clear that classes are just one of the many possible ways of generating objects with common properties", and that whereas "class-based languages integrate many ideas into a single construct, i.e. the class", "object-based languages decompose class-based features" into fundamental parts, in order to "reconstruct them in different ways". The bulk of Abadi and Cardelli's book is taken up in defining a family of *object calculi*, starting with an untyped calculus, and proceeding through typed first and second-order calculi, to a typed higher-order calculus.

In the untyped calculus, an object o is defined as a collection of components $l_i = \sigma(x_i)b_i$ for $i \in 1..n$, which is denoted by

$$[l_i = \sigma(x_i)b_i^{\,i\in(1..n)}] \ ,$$

where l_i is a distinct label, $\sigma(x_i)b_i$ is an associated method, and x_i is a variable that binds the body b_i of the method to *self*. An invocation of method l on object o is denoted by $o.l$, and an update of method l on object o with method $\sigma(x)b$ is denoted by $o.l \hookleftarrow \sigma(x)b$. The execution of a particular term can be expressed as a sequence of reductions, where the one-step reduction rules (method invocation and method update) are given by

$$\begin{aligned} o.l_j &\rightarrowtail b_j\,[o/x_j] \\ o.l \hookleftarrow \sigma(x)b &\rightarrowtail [l_j = \sigma(y)b, l_i = \sigma(x_i)b_i^{\,i\in(1..n)-\{j\}}] \ , \end{aligned}$$

for $j \in 1..n$.

In the first-order calculus, an object is defined in much the same way as in the untyped calculus, except that its components l_i and bound variables x_i are typed. The proofs of two properties of the first-order calculus are given, namely that if a term has a type, then that type is unique; and the steps associated with the reduction of terms of the untyped calculus extend to those of the typed calculus. In the second order calculus, three forms of quantification are introduced: universal, existential and *self* quantification, the last one being a combination of recursion and bounded existential quantification. Finally, in the high-order calculus, the first order calculus is enriched with, amongst other things, a structural rule for method update, which ultimately leads to an object-oriented counterpart of $F_{w<:}$ [40] with recursive types, namely $Ob_{w<:\mu}$.

Fisher, Honsell and Mitchell [56] present a typed calculus of functions and objects as a means of studying the typing of object-oriented languages, in particular of inherited methods. The calculus is based on an extended form of the untyped lambda calculus, with additional object-related terms for: $\langle\rangle$, the empty object; $e \Leftarrow m$, the operation 'send message m to object e'; $\langle e_1 \leftarrow\!+\, m = e_2\rangle$, the object obtained by extending e_1 with a new method m having body e_2; and $\langle e_1 \leftarrow m = e_2\rangle$, the object obtained by replacing e_1's

[1] The former includes the notions of object, class, subclass, subsumption, method replacement (overriding), method reuse (inheritance) and method specialisation (flexible overriding); the latter includes the notions of objects without classes, prototypes and clones, and inheritance by both embedding and delegation.

method body for m with e_2. The typing rules associated with the extension and replacement operations are quite different, because if an object is extended with a new method, no other method could possibly refer to it, and therefore no account needs to be made of its type. However, if a method is replaced by one with a different type, care must be taken to avoid violating the types of terms in other methods that refer to it. Once the typing rules are given, the bulk of what remains is taken up in proving that the type system is sound, i.e. if $\Gamma \vdash e\colon \tau$ is derivable, then $\mathit{eval}(e) \neq \mathit{error}$, where *eval* is an evaluation strategy, and *error* is a special symbol returned by the evaluator to indicate that it has failed to find a method. The proof justifies the author's claim that the type system prevents "message not understood" errors.

7.1.6 Inheritance

Kamin and Reddy [75] compare the semantics of two models of object-oriented languages—the *closure* model and the *data structure* model—by developing for each model the semantics of a series of small abstract languages: *ObjectTalk*, a language with objects and operations but no classes; *ClassTalk*, a language with objects as instances of classes sharing a common set of operations; *InheritTalk*, a language with classes inheriting from other classes; and *SmallTalk*, a language with inheritance in the style of *SMALLTALK-80*. In the closure model, closures are used to encapsulate the side effects on objects, and consequently the operations on an object are defined to be part of the object. In the data structure model, objects are represented by records of instance variables, and the operations on objects are defined separately. Kamin and Reddy prove that the two models of *ObjectTalk* are semantically equivalent.

Cook, Hill and Canning [41] eschew the notion that "inheritance is subtyping" by introducing a new system of inheritance based on types, composed of an extended polymorphic λ-calculus and a denotational model of inheritance, which supports the incremental extension of recursive structures at three levels: object, class and type. At the *object* level, the model supports the incremental construction of objects, where the type of an object formed by inheritance is considered to be different from the type of the object from which it inherits. At the *class* level, the model supports the incremental definition of classes, where the inherited constructors are adapted to create objects of the inherited class. Finally, at the *type* level, the model supports the definition of recursive types, where new recursive record types are formed by extending existing recursive structures with additional fields. However, unlike Cardelli and Mitchell's model [34], in which record types are the kinds of records that have *at least* a specified set of fields, the record types of Cook, Hill and Canning's model are the kinds of records that have an *exact* set of fields; this makes their model simpler.

Mitchell [90] discusses method specialisation in the context of object oriented programming languages, and introduces a typed calculus of objects and classes in which methods can be added or overridden. An example of the latter is the method *move* defined on a class of points, i.e.

$$\textbf{class}\ \mathit{point}\ \textbf{methods}\ x\colon \mathit{int},\ y\colon \mathit{int},\ \mathit{move}\colon \mathit{int} \times \mathit{int} \to \mathit{point}\ ,$$

which is overridden by a method of the same name defined on a specialised class of coloured

points, i.e.

$$\textbf{class } \mathit{coloured} \textbf{ methods } x\colon \mathit{int},\ y\colon \mathit{int},\ c\colon \mathit{color},\ \mathit{move}\colon \mathit{int} \times \mathit{int} \to \mathit{coloured}\ .$$

When the *move* method is specialised, its type clearly changes. Mitchell notes that while this behaviour is familiar to a programmer of an object oriented language, it is nonetheless difficult to simulate in a traditional language like Pascal or Ada. Method specialisation is achieved by treating objects as collections of functions, each representing a method of the object. When a method is invoked, the appropriate function is applied to the object using the method's first argument, rather than a special symbol like *self*. The soundness of the typing rules Mitchell introduces is suggested by a translation to a traditional calculus with recursively defined record types.

7.2 Rewriting Logic

This section reviews a selection of related work in the field of rewriting logic, which has its origins in the work of Meseguer [86, 87].

Meseguer [86] introduces rewriting logic as a logic that is ideally suited to the task of capturing the semantics of concurrent object-oriented computations, in a declarative style of programming where computation corresponds exactly to logical deduction. A theory in rewriting logic is a tuple (Σ, E, R), where (Σ, E) is an equational theory for representing the static aspects, and R is a set of possibly conditional rewrite rules for representing the dynamic aspects, of a concurrent or logic system. Meseguer suggests that the (static) distributed states and (dynamic) transitions of a concurrent system, are closely allied to the (static) formulas and the (dynamic) inferences of a logic system, the suggestion being that they are essentially two sides of the same coin. Meseguer also introduces a new programming language called *Maude* [39], which is directly based on rewriting logic, and which uses OBJ3 [59] as a sublanguage. Maude supports two kinds of modules: *functional* and *system*. In functional modules, computation proceeds by performing equational simplification from left to write using `eq` declarations, until no more simplifications are possible. In system modules, where `eq` declarations (equations) are replaced by `rl` declarations (rules), different semantics apply and the traditional interpretation of rewrite rules as equations is abandoned.

In object-oriented Maude, the statement

```
class C | a1 : s1, ..., an : sn
```

declares a class `C` with attributes `a1` to `an` of sorts `s1` to `sn`; and the statement

```
< O : C | a1 : v1, ..., an : vn >
```

declares an object `O` of class `C` with attribute values `v1` to `vn`. Further, the dynamic behaviour of a concurrent system of objects is specified by means of rewrite rules, like

```
rl [l] : < O : C | a1 : 0, a2 : y, a3 : w > =>
         < O : C | a1 : v, a2 : y, a3 : y + w > ,
```

which defines a family of transitions between `O` and itself, that can be applied whenever `a1` has the value `0`. In Maude, a subclass inherits all of the attributes and rewrite rules of its superclasses.

Boronat and Meseguer [23] propose an algebraic semantics for a subset of the OMG's metamodelling framework MOF [97], based on a combination of membership equational logic (MEL) [88] and rewriting logic. In particular, they define a formal semantics for three important concepts: *metamodel*, *model*, and *conformance* (that is of models to metamodels), of which none is adequately defined by the MOF. The need for a formal semantics for the MOF is particularly important because so many modelling languages depend on it. Boronat and Meseguer's work provided the foundational framework for a model management tool called MOMENT2 [22] based on Maude, which opened up the possibility of developing a wide range of model-driven applications in a formal setting, including metamodel conformance checkers, QVT-like model transformations, and domain-specific language specifiers. MEL theories and rewrite theories are specified using functional modules and system modules respectively.

The Architecture Analysis and Design Language (AADL) [53] supports the early analysis of a system's architecture in terms of its performance-critical properties. It is especially effective at analysing the behaviour of complex, real-time embedded systems. Ölveczky, Boronat, Meseguer and Pek [104] define a formal semantics in rewriting logic for a substantial subset of AADL, to address a weakness in the AADL standard, namely its failure to define the semantics of AADL with sufficient rigour, leaving it open to interpretation. Their semantics are formalised in real-time Maude [105], an extension of the Maude specification language and its underlying formalism. A tool for automating the translation of AADL models in OSATE (an open-source AADL tool[2]) to real-time Maude specifications is also described. Under normal circumstances, AADL models are not executable. However, by translating them into Maude specifications, they become amenable to simulation, reachability analysis and model checking. Further, because the representational distance between AADL models and their formal specifications is small, it is relatively easy to map—for diagnostic purposes—analysis results back to the AADL models from which they were derived.

A large software system comprises a wide variety of artefacts: requirement specifications, design models, executable code, test suites, configuration files and so on. Each artefact describes the system from a particular point of view and level of abstraction. As systems evolve, it is extremely challenging to manage the unintended consequences of changing one artefact with respect to another. As Eqyed recalls [50], "changes are inevitable during software development, and so are their unintentional side effects". Boronat and Meseguer [24] describe a technique, based on rewriting logic, for finding and resolving inconsistencies in MOF based, heterogeneous specifications, i.e. specifications in which the models are not all conformant to the same metamodel. In their approach, inconsistencies are formally defined as propositions in equational logic, and once detected, Maude's pattern matching algorithms are used to find an efficient repair plan, to automatically bring a heterogeneous specification to a state that minimises the number of inconsistencies. Boronat and Meseguer's technique

[2]See `https://wiki.sei.cmu.edu/aadl/index.php/Osate_2`.

is illustrated with a UML case study adapted from Egyed, Letier and Finkelstein [51].

7.3 Type Theory

This section reviews a selection of related work with a strong type theoretical foundation.

The use of type theory to implement model transformations has its roots in the work of Poernomo [110][3], who proposes a type-theoretic framework for formalising the OMG's Meta Object Facility (MOF) [99], as a foundation for improving the trustworthiness of model driven engineering. As described earlier, the MOF is a layered structure for managing metadata. At level M_0, there are instances of model classes. At level M_1, there are dual representations of models, as classes and as instances of metamodel classes. Similarly at level M_2, there are dual representations of metamodels, as classes and as instances of metametamodel classes. Finally, at level M_3, there are the metametamodel classes. Poernomo associates the levels of the MOF with a predicate hierarchy of type universes, as shown in Figure 7.1, in which the classes of the metametamodel at M_3 (e.g. $Metaclass$) are

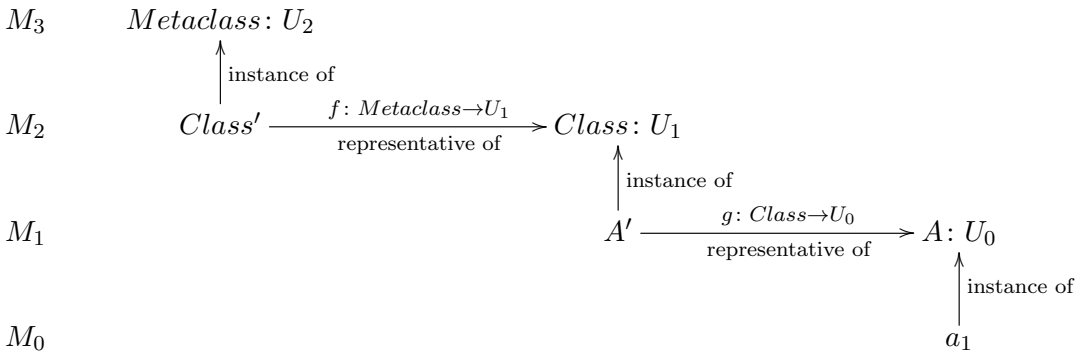

Figure 7.1: The MOF as a predicative hierarchy of type universes.

types in universe U_2, the classes of metamodels at M_2 (e.g. $Class$) are types in universe U_1, and the classes of models at M_1 (e.g. A) are types in universe U_0. At M_2, there is a function which maps the instances of metametamodel classes (e.g. $Class'$) to their representations as classes (e.g. $Class$). Similarly, at M_1, there is a function which maps the instances of metamodel classes (e.g. A') to their representations as classes (e.g. A). A class-based model at M_1 is said to *conform* to a class-based metamodel K at M_2, if its instance-based representation at M_1 inhabits K. Similarly for a class-based metamodel at M_2.

In [112], Poernomo extends these ideas to model transformations, by showing how the proofs-as-programs paradigm can be used to extract provably correct model transformations from specifications of transformations as types of the form

$$\forall x\colon PIL\,.\,I\,x \to \exists y\colon Y\,.\,O\,x\,y\ ,$$

where PIL and PSL are the source and target metamodel types, I is a precondition on source metamodels x, and O is a postcondition on source and target metamodels x and y.

[3] A shortened version of this paper may be found in [111].

Calegari [31] experiments with a type-theoretic approach to the verification of model transformations in the Calculus of Inductive Constructions (CIC) [42, 132], where a model is represented as an inductive type (containing lists of instances of metamodel classes, like Poernomo's instance-based representation of a model in [110]), and a metamodel is represented as a coinductive type, irrespective of the kinds of relations between metaclasses. Calegari shows, by means of a concrete example taken from an online repository of ATL transformations [2], how parts of the ATL language can be represented in the CIC, including matched rules, helpers and expressions based on the Object Constraint Language (OCL) [102]. Like other authors in this field, Calegari formalises the specifications of model transformations as $\forall\exists$ formulas.

In a later paper, Calegari [30] proposes a fundamentally different way of formalising metamodels, for reasons that will be explained by example. Consider a model in which R is a relation between classes A and B. There are essentially two ways of formalising R: first, as an inductive type with references to A and B, where A and B are formalised as inductive types too (call this the *external* approach); second, as a reference in A to B and/or a reference in B to A, where A and B are formalised as either inductive or coinductive types (call this the *internal* approach). Now, no matter what the conditionality and directionality of R, the external approach is always available. However, with the internal approach, there are a number of cases to consider. First, if R is a unidirectional relation, an inductive formalisation of A and B is available. Second, if R is a bidirectional and unconditional relation, a coinductive formalisation of A and B is a *necessity*. Third, if R is a bidirectional and conditional relation, a mutually inductive formalisation of A and B is available. The point is that dependencies between classes need not result in dependencies between objects. In this later paper, Calegrai eschews coinductive types completely because—as he sees it—they compromise the correctness of a model. With regards the second internal case, Calegari uses external formalisations to break the cycles of dependency between objects. Of course, this discussion extends to any number of classes; it is not restricted to two.

7.4 Graph Theory

This section reviews a selection of related work with a strong graph theoretical foundation, by the most prominent researchers in the field.

VIATRA2 [135] is a unidirectional, hybrid model transformation language, which was founded on two mathematically precise formalisms: graph transformation [52] and abstract state machines [21]. It comprises three coherent sublanguages.

- The *Visual Textual Metamodelling Language* (VTML) is a language for describing metamodels. Unlike most of its counterparts, though, it is not based on the OMG's Meta Object Facility (MOF) [99] but rather on the Visual and Precise Metamodelling (VPM) facility [136], because in the eyes of the authors of VIATRA2, the MOF fails to adequately describe how "metamodel levels are created and related to one another" [11]. A simple VTML script of a relation `attr` between two classes `Class` and `Attribute` in a metamodel `UML`, is shown below.

```
entity(UML)
{
   entity(Class);
   entity(Attribute);
   relation(attrs, Class, Attribute);
}
```

- The *Pattern Language* is a language for specifying common patterns in transformation rules. Patterns are like predicates in Prolog. For example, the following pattern is fulfilled if `A` is an attribute of `C`.

```
pattern isClassAttribute(C, A)
{
   Class(C);
   Attribute(A);
   Class.attrs(X, C, A);
}
```

  A pattern can call other patterns; it can also call itself.

- The *Transformation Language* is a language for manipulating models. It uses graph patterns to define graph transformation rules. A graph transformation rule contains a precondition pattern for the left hand side `lhs` and a postcondition pattern for the right hand side `rhs`, i.e.

```
gtrule XYZ(...)
{
   precondition pattern lhs(...) = {...}
   postcondition pattern rhs(...) = {...}
}
```

  In general, (model) elements that are only present in an image of the `lhs` are deleted; elements that are only present in an image of the `rhs` are created; and all other elements remain the same. Rules are used to drive the execution of a transformation, e.g.

```
rule main(...)
{
   forall ... do XYZ(...);
}
```

The theory of graphs [47] is a well established field of study and research, and is employed on a wide range of applications that have a need to store structured collections of related artefacts. For example, Blostein [18] describes an application of graph transformation to the interpretation of mathematical expressions, where symbols are represented as the vertices of a graph with "meaning" and "location" attributes. The graph is initially

edge-less, but after the application of a series of graph rewrite rules, first to build the links between symbols based on their locations (e.g. a "is to the left of" b in the expression ab), second to constrain the links to remove redundancy, and third to parse the resulting graph, what emerges is a solitary vertex whose "meaning" attribute describes the entire mathematical expression. An example closer to home is provided by Agrawal [10], who developed a prototypical transformation language and a graph rewrite engine (GRE), for transforming platform independent models into platform specific models. The GRE takes as input a source metamodel, a target metamodel, a transformation specification, and a model conforming to the source metamodel, and produces as output a model conforming to the target metamodel. A transformation is specified by a sequence of rules, where a rule defines a relationship between a subgraph of the source metamodel (known as the pattern or LHS), a subgraph of the target metamodel (known as the RHS), and a set of actions to be performed if a subgraph of the source model matches the pattern. The GRE evolved into the Generic Modelling Environment [9], and then the Graph Rewriting and Transformation Language (GReAT) [13].

In [29], Bruni puts forward the case for exploiting the hierarchical structure of models, in domains as diverse as software architectures and business processes, where hierarchy is an inherent part of the fabric of a domain. In this setting, a model is understood to be a particular kind of hierarchical graph, a configuration of objects whose attributes are either properties of objects or references to related objects. The author notes that "a prominent type of relation among objects is structural containment", and proceeds to introduce the notion of a nested collection of objects as a kind of attribute in its own right, which is not that far removed from the type theoretical formalisation of a many-valued relation given earlier. Interestingly, the author goes on to describe how a hierarchical model transformation of an arbitrary depth can be inductively defined by a set of transformation rules in the style of Plotkin [109]. Bruni [28] lays down what he sees as the ten virtues of structured hierarchical graphs over their unstructured flat counterparts, while Drewes [48] addresses the need to improve the comprehensibility of large graph transformations using rule-based transformations of hierarchically structured hypergraphs, in which a hypergraph contains hyperedges which contain further hypergraphs and so on.

The use of triple graph grammars (TGGs) to declaratively specify bidirectional model transformations has its origins in the work of Schürr [127], who invented the concept of TGGs as a generalisation of Pratt's earlier work on pair grammars [114]. A TGG is a grammar that produces a language of graph triples $\langle LG, CG, RG \rangle$, where LG is the left or source graph, RG is the right or target graph, and CG is a correspondence graph in between, which captures the relations between the vertices of LG and RG. A TGG can be compiled into both a forward translator that takes LG to RG, and a backward translator that takes RG to LG. In [60], Golas extends the theory of model transformations based on TGGs to rules with application conditions—conditions under which the rules can be applied—in order to strengthen the expressive power of model transformations. A condition is said to be positive if it *demands* the existence of a particular structure, and negative if it *forbids* it. Finally, Grunske [64] describes the development of a tool which transforms TGGs into productions defined by the Tefkat [5] model transformation language, using the Fujaba Tool Suite [6] and a TGG plugin.

7.5 Category Theory

This section reviews a selection of related work with a strong category theoretical foundation. The reader should note that Fiadeiro [55] and Pierce [107] provide good introductory texts on category theory, specifically aimed at software engineers and computer scientists.

Schulz, Löwe and König [125] present a categorical framework for refactoring object-oriented models and data, which supports the co-evolution of model transformations and instance migration (i.e. the derivation of new instance data from old instance data). As Fowler [57] puts it, "refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure". Typical examples of object-oriented refactorings are "adding a new class A to a model and then making an existing class B a subclass of A", and "moving the origin of an association from a subclass B to a superclass A". The combined effect of these particular refactorings might be to prepare a model for future upgrades, by allowing new subclasses of A to be easily slotted in alongside B. A transformation $t\colon S \rightsquigarrow S'$ in the category of model parts, where a part is either a class in a model or an item of instance data, and an arrow between parts is either an association or a link, is defined by the span

$$S \xleftarrow{l} S^{\#} \xrightarrow{r} S'$$

for some part $S^{\#}$. The span defines the relationship between the old and new parts S and S' respectively. In general, a transformation allows parts to be reduced and extended through non-surjective morphisms l on the left, and folded and unfolded through non-injective morphisms r on the right. However, a refactoring kind of transformation is only allowed to perform surjective morphisms on the left, to prevent it from deleting data and losing information. The innovative part of this work is in its use of categories and functors to simultaneously transform models and data.

Minas and Schneider [89] follow Rydeheard and Burstall [124] in building a bridge between category theory and computer programming, by showing how the basic constructions of category theory—objects, morphisms, categories, colimits and so on—can be implemented in Java. A category of graphs built out of these constructions yields a simple and modular implementation of graphs and graph transformations. A category—essentially a class of objects and a class of morphisms between objects that satisfy certain properties—is specified in Java by a pair of generic interfaces, parameterised by the types of object `O` and morphism `M`, i.e.

```
public interface Mor<O, M> {...}
public interface Cat<O, M extends Mor<O, M>> {...} .
```

The interfaces for a particular kind of object are realised by concrete classes, e.g.

```
public class SetMor implements Mor<Set<?>, SetMor> {...}
public class SetCat implements Cat<Set<?>, SetMor> {...} ,
```

where `Set<?>` is the type of a set of arbitrary Java objects. A category of graphs is easily specified by a pair of interfaces too, i.e.

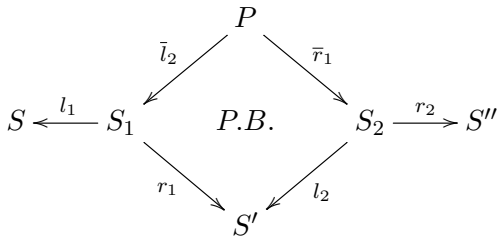

Figure 7.2: The composition of transformations $S \rightsquigarrow S'$ and $S' \rightsquigarrow S''$.

```
public interface Graph<S, F> {...}
public interface GraphMor<F> {...} ,
```

where `S` and `F` could be bound to `Set<?>` and `SetMor` in any realisation of the category. A graph transformation, which is implemented by the single pushout (SPO) approach, is based on partial graph morphisms. A partial graph morphism between graphs `G` and `H` is a structure-preserving pair of partial functions, which take the nodes and edges of `G` to `H`.

Schulz, Löwe and König [126] take their earlier work a stage further by investigating the circumstances under which composed model transformations can be shown to be compatible with composed instance migrations. They propose a formal method based on universal algebra and category theory, for simultaneously describing models and instance data at MOF levels M1 and M0 respectively. Using sound categorically-founded constructions, the formal method guarantees that the migrated instance data is adapted as much as necessary, and as little as possible. The composition of two model transformations

$$S \xleftarrow{l_1} S_1 \xrightarrow{r_1} S'$$

and

$$S' \xleftarrow{l_2} S_2 \xrightarrow{r_2} S'' ,$$

is defined to be the span

$$S \xleftarrow{l_1 \circ \bar{l}_2} P \xrightarrow{r_2 \circ \bar{r}_1} S'' ,$$

where

$$S_1 \xleftarrow{\bar{l}_2} P \xrightarrow{\bar{r}_1} S_2$$

is the pullback of

$$S_1 \xrightarrow{r_1} S' \xleftarrow{l_2} S_2 ,$$

see Figure 7.2. The composition of instance migrations is shown to be compatible with the composition of model transformations if a certain commutativity property holds between two functors.

7.6 Other Work

This section reviews a selection of other related work.

The Atlas Transformation Language (ATL) [73, 72] is a hybrid language which supports both declarative and imperative forms of specification. In its declarative form, an ATL transformation is a conditional mapping between M source metamodels and N target metamodels, which is specificed by a set of *rules*—possibly aided by so-called *helpers* that encapsulate imperative statements written in the OMG's Object Constraint Language [102]—where a rule defines a correspondence between a source model element and a target model element. For example, the following rule defines the specification of a transformation between the persistent classes `c` of a source model conforming to the `Uml` metamodel, and the tables `t` of a target model conforming to the `Sql` metamodel, in which tables have the same names as classes.

```
rule Uml2Sql {
   from
      c : Uml!Class (c.is_persistent)
   to
      t : Sql!Table (name <- c.name)
}
```

ATL transformations are not guaranteed to terminate, because there is nothing to preclude a helper from looping indefinitely. The ATL tool suite [71] provides good support for developing model transformations.

The Query/View/Transformation (QVT) language [63] was discussed earlier in Section 3.2.5. In [78], Kurtev provides a nice overview of QVT, and draws attention to a number of important issues regarding model transformations, including the need to preserve the meanings of models as they pass through what could be several levels of refinement.

Jouault [74] compares and contrasts the major features of ATL and QVT, and a number of other *dedicated* model transformation languages besides, including the graph transformation language VIATRA2 which is described in the previous section. A number of general purpose languages are also used to implement model transformations, type theory being an obvious example in this context, although not a very popular one, because the effort required to implement even the most modest of transformations is considerable. In industry, the focus is primarily on developing model to text transformations using proprietary solutions: see Abstract Solutions' TA-5M Code Generator [1] and IBM's Rational Rhapsody Developer [3].

The *by example* paradigm, in which abstract problems are solved using concrete examples, dates back to the 1970s when Zloof [142] proposed a language for querying databases by example, which allowed non-specialist users to acquire the necessary skills to make quite complex database queries after only a few hours tuition. Many years later, Liebermann [81] devised a teaching aid which enabled novice programmers to construct Lisp programs by example, from concrete input parameters and step by step instructions on how they should be processed.[4] There are many other examples in the early literature. More recently,

[4] A trivial example of such a program is given by Halbert [65]. Declare a one-place function, `second` say,

Varró [134] proposed a framework for transforming models by example, to enable modellers of problem domains (who may be experts in their own fields but who have little knowledge of metamodelling) to develop model transformations—at least in part—in languages with which they are most familiar. Varró's iterative approach, which is comparable to the one described by Wimmer [140], can be summarised as follows. First, pairs of interrelated source and target models are assembled to bring out the critical characteristics of the transformation under development. For example, if the source model is hierarchical and the transformation of a source class is dependent on its position in the hierarchy, several pairs of models would be required to cover all possible cases. Second, the paired models are automatically transformed into a set of rules—in the case of Wimmer, rules of the Atlas Transformation Language (ATL) [72]—which correctly transform the source models from which they were derived. Third, the transformation rules are manually refined as necessary. Fourth, the transformation rules are executed on all source models and validated against their respective target models. One benefit of the model transformation by example (MTBE) approach is that the cost of developing a test suite is minimal, since it can be heavily based on the pairs of source and target models created in the first step. Finally, new pairs of source and target models are assembled and new transformation rules are generated if further tests are deemed necessary. Balogh [14] proposes the use of inductive logic programming [93] to automate Varró's MTBE approach.

Kolovos [77] shows how the process of testing model transformations can benefit from the automation of another model management task, i.e. model comparison. Two models l and r are compared by partitioning their elements into a number of categories, including "elements of l that match those of r" according to some criteria (e.g. class names are the same), and subcategory "elements that match but do not conform" according to some criteria (e.g. one class is abstract, the other is not). If l and r are the source and target models of a transformation, it may be possible to infer from the categories of a comparison of l and r, which elements have not been transformed correctly, e.g. the elements that match but do not conform are likely candidates.

Hemel [66] describes an interesting case study of "code generation by model to model transformations", in a language that was designed for the specification of transformation systems based on the paradigm of rewriting strategies. In Stratego [137], *rewrite rules* are used to define elementary transformations between fragments of the source language on the left, and fragments of the target language on the right; and *strategies* are used to control the execution of assemblies of rewrite rules in large transformations. Moreover, generated code has a structured representation in Stratego, which is presumably how it can support sophisticated code to code transformations, as well as model to model, and model to code transformations.

---

that returns the second element of a list. Provide a concrete example of an input parameter, `(1 2 3)` say. Apply `cdr`. The system returns `(2 3)`. Apply `car`. The system returns `2`. Declare the function complete. The system returns `(defun second (a) (car (cdr a)))`. Clearly, care must be taken in choosing examples that are representative of the universe of possible input parameters, otherwise programs may produce unexpected results. Indeed, `second` would almost certainly fail if it were invoked with an empty list or a one-element list, and therefore in addition to choosing representative examples, one must also choose enough examples to cover all possible cases.

Lano [79] uses the B language [8] to verify the semantic properties of UML-RSDS models [80] and the correctness of model transformations. UML-RSDS is a subset of UML with precise semantics. UML-RSDS models are translated into B for semantic analysis, in particular for showing—by the B concept of formal refinement— that the properties of a model before and after it is transformed are preserved. Consider a transformation between two UML-RSDS models. Let σ be the source model and let B_σ be the translation of σ. Similarly, let τ be the target model and let B_τ be the translation of τ, where B_τ is defined to be a refinement of B_σ as part of the translation process. From B_τ, proof obligations are generated which when proved show that the pre and post properties of operations, the static invariants, and the possible initialisations in σ, are also valid in τ.

8 Conclusions

In conclusion, this chapter summarises the thesis, revisits the contributions that the author claims to have made in the field of study to see if they are justified, and comments on the outlook for future work.

8.1 Summary

Apart from the introduction, related work and conclusions chapters, this thesis contains three background chapters (on type theory, models and model transformations), and two foreground chapters (on ordered model transformations and certified ordered model transformation), the second of which culminates in the author's main contribution, the logical interpretation theorem. The two foreground chapters are analogues of each other: one describes *uncertified* ordered model transformations, the other describes *certified* ordered model transformations. The author initially baulked at the idea of writing two such similar chapters. However, once the supplementary information surrounding the two sets of data types had been added—the logical interpretations of the components in the former, the logical interpretation theorem in the latter—it became clear that the decision to define the certified components anew, rather than as appendages of the uncertified components, was the right decision.

The principle purpose of this thesis is to show that the proof of an arbitrarily large ordered model transformation is logically equivalent to the sum of the proofs of its component parts. By way of analogy, if one associates the proof of an ordered model transformation with a jigsaw puzzle, which is of a size and shape commensurate with the size and shape of the transformation, and identifies each piece of the jigsaw with a fragment of the proof, then clearly every piece is coupled with either a component proof, or with the "glue" that binds the component proofs together. Now, if one removes the *transformation-specific* component proofs from the puzzle, what remains is a partially completed *transformation-independent* proof "with holes", where the holes are capable of being filled by the component proofs of *any* provable ordered model transformation of the same size and shape. The logical interpretation theorem affirms that the proof of an ordered model transformation (if indeed it is provable) need not be constructed in its entirety.

It became clear from the outset that there was scope to factor out parts of the proofs of ordered model transformations; one only has to look at the proofs of a graded set of examples to see sequences of inference rules repeating themselves over and over again. However,

it took quite a long time for a suitable scheme to emerge. Initially, there were no component types, just one large monolithic type which captured everything. Needless to say, it was impossible to describe. The decisions to decompose an ordered model transformation into a set of component types, and decouple the structure of an ordered model transformation from its logical interpretation (with the latter derived from the former by means of a recursive function), proved crucial. However, while this was definitely the right thing to do, it nevertheless proved problematic from a timing point of view because it required the development in short order of a large number of packing and unpacking functions to support the specification and proof of the logical interpretation theorem.

Of the other challenges the author faced, the one that stands out most concerns presentational rather than technical matters, in particular how best to present the proof of the logical interpretation theorem. The author's motive in choosing to play out almost every step of the proof by hand was purely pedagogical, having found some aspects of type theory (in particular, the elimination and computation rules of dependent types) difficult to understand, and assumed that others may find them difficult too.

8.2 Contributions

The contributions that the author claims to have made in the field of model transformations are justified as follows.

1. *Devised a method for proving the correctness of an arbitrarily large ordered model transformation, by showing how its proof can be assured by summing the proofs of its parts, thereby making it easier to derive.*

   This claim is justified in virtue of Theorem 4, the logical interpretation theorem, and the related examples in Sections 5.4 and 6.4.

2. *Raised the prospect of decomposing a class of transformations into a set of horizontal and vertical components, where each horizontal component is responsible for transforming data between the source and target models, and each vertical component is responsible for transforming links.*

   This claim is rather nebulous and difficult to assess. The prospect has certainly been raised that some kinds of transformations—not only ordered model transformations—may benefit from a two dimensional approach, whereby the transformation of (horizontal) attributes is treated separately from the transformation of (vertical) relations, but that is about as far as it goes.

3. *Formulated a means of defining recursive specifications of model transformations in type theory.*

   This claim is justified in virtue of functions *Spec* in Section 4.5 and Log_{Tran} in Section 5.3.2. However, while it could be construed as obvious, with the benefit of hindsight, that a recursive function is required to define a recursive specification, it was not at all obvious how to realise such a function in practice, for there were no precedents for handling recursive specifications in this field.

4. *Composed a type system for implementing model transformations.*

   This claim is justified in virtue of the existence of the type system in Section 2.6, and the fact that it has been used to develop a number of model transformations. However, in the light of some hard lessons learnt about particular aspects of type theory, the author's original aim—to present the material in a tutorial manner—was not achieved due to lack of time. On reflection, the aim was probably neither appropriate nor achievable. Nevertheless, the type system that remains stands as a reasonable reference for anyone wishing to implement model transformations in constructive type theory.

8.3 Outlook

The author is under no illusions that this thesis is going to change the face of model transformations. On the contrary, it features a particular kind of model transformation which is so limited in scope as to be of little use in the real world without significant enhancements. Nevertheless, there is something interesting going on here which merits further study. As remarked earlier, the author develops large model transformations in industry, and although the models are not strictly ordered by containment, many of them have a strong hierarchical core around which transformations are invariably based. The imperative algorithms which drive the transformations always start at the root class and step down the containment and generalisation hierarchies, transforming classes along the way and only deviating off line to pick up extra data.

A suitable next step would be to apply some of these ideas to the development of a certified executable UML to Java or C++ translator. Such a product would be an enormous asset to any organisation developing UML based software for high integrity systems.

Lastly, in virtue of the similarity between class diagrams and category diagrams, it has been suggested that the theory of ordered model transformations may have benefited from a category theoretical, rather than a type theoretical, approach, in which the models are represented as categories, and the transformations are represented as functors. This should be investigated as part of any future work.

Bibliography

[1] Abstract Solutions TA-5M Code Generator. `http://www.abstractsolutions.co.uk/PRODUCTS/iccg/cppcode.php`.

[2] ATL Basic Examples and Patterns. `http://www.eclipse.org/atl/documentation/basicExamples_Patterns/`.

[3] IBM Rational Rhapsody Developer. `http://www-03.ibm.com/software/products/us/en/ratirhap`.

[4] Object Management Group. `http://www.omg.org`.

[5] Tefkat Model Transformation Language. http://tefkat.sourceforge.net.

[6] The Fujaba Tool Suite. http://www.fujaba.de.

[7] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[8] J. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[9] A. Agrawal, G. Karsai, and A. Ledeczi. An End-to-End Domain-Driven Software Development Framework. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA 2003, pages 8–15, 2003.

[10] A. Agrawal, T. Levendovszky, J. Sprinkle, S. F., and G. Karsai. Generative Programming via Graph Transformations in the Model-Driven Architecture. In *Workshop on Generative Techniques in the Context of Model Driven Architecture*, OOPSLA 2002, 2002.

[11] C. Atkinson and T. Kühne. The Essence of Multilevel Metamodelling. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *LNCS*, pages 19–33. Springer, 2001.

[12] P. Bagely. *Extension of Programming Language Concepts*. University City Science Center, Philadelphia, 1968.

[13] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The Graph Rewriting and Transformation Language : GReAT. In A. Zündorf and D. Varró, editors, *Proceedings of the Third International Workshop on Graph Based Tools*, volume 1 of *Electronic Communications of the EASST*, pages 1–8, 2006.

[14] Z. Balogh and D. Varró. Model transformation by example using inductive logic programming. *Journal of Software Systems Modelling*, 8(3):347–364, 2008.

[15] M. Beeson. Foreword to Foundations of Constructive Analysis by Erret Bishop, 2012.

[16] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[17] E. Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, New York, 1967.

[18] D. Blostein and A. Schürr. Computing with Graphs and Graph Transformations. *Software - Practice and Experience*, 29(3):197–217, 1999.

[19] G. Booch. *Object Oriented Analysis and Design*. Addison-Wesley, 1994.

[20] G. Booch, J. Rumbaugh, and I. Jaconson. *The Unified Modelling Language User Guide*. Addison-Wesley, 1998.

[21] E. Börger and R. Stärk. *Abstract State Machines. A Mtehod for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[22] A. Boronat. *MOMENT: A formal framework for MOdel manageMENT*. PhD thesis, Universitat Politènica de València (UPV), Spain, 2007.

[23] A. Boronat and J. Meseguer. An algebraic semantics for MOF. In *Fundamental Approaches to Software Engineering*. Springer, 2008.

[24] A. Boronat and J. Meseguer. Automated Model Synchronization: A Case Study on UML with Maude. *European Joint Conferences on Theory and Practice of Software*, 2, 2011.

[25] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as Implicit Coercion. *Information and Computation*, 93:172–221, 1991.

[26] L. E. J. Brouwer. On the Foundations of Mathematics. *Collected Works*, 1:11–101, 1907.

[27] K. B. Bruce and G. Longo. A Modest Model of Records, Inheritance, and Bounded Quantification. *Information and Computation*, 87:196–240, 1990.

[28] R. Bruni and A. Lafuente. Ten virtues of structured graphs. In A. Boronat and R. Heckel, editors, *Graph Transformation and Visual Modeling Techniques*, volume 18 of *Electronic Communications of the EASST*, pages 1–20, 2009.

[29] R. Bruni, A. Lafuente, and U. Montanari. On Structured Model-Driven Transformations. *International Journal of Software and Informatics*, 2:185–206, 2011.

[30] D. Calegari, C. Luna, N. Szasz, and A. Tasistro. A Type-Theoretic Framework for Certified Model Transformations. In *Proceedings of the 13th Brazilian conference on Formal methods: foundations and applications*, SBMF'10, pages 112–127. Springer-Verlag, 2010.

[31] D. Calegari, N. Luna, C. Szasz, and A. Tasistro. Experiment with a Type-Theoretic Approach to the Verification of Model Transformations. In *Proceedings of the Second Chilean Workshop on Formal Methods*, pages 29–36, 2009.

[32] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, number 173 in LNCS. Springer-Verlag, 1984.

[33] L. Cardelli. Extensible Records in a Pure Calculus of Subtyping. Technical report, Digital Equipment Corporation, Systems Research, 1992.

[34] L. Cardelli and J. C. Mitchell. Operations on Records. *Mathematical Structures in Computer Science*, 1:3–48, 1991.

[35] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:47–522, 1985.

[36] A. Church. The Calculi of Lambda-Conversion. In E. Artin and M. Mores, editors, *Annals of Mathematics Studies*, number 6. Princeton University Press.

[37] A. Church. A set of postulates for the foundation of logic. *Annals of mathematics*, 33:344–366, 1932.

[38] A. Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2):56–68, Jun. 1940.

[39] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J. Quesada. A maude tutorial. *Computer Science Laboratory, SRI International*, 2000.

[40] A. B. Compagnoni. *Higher-order subtyping with intersection types*. PhD thesis, CipData, Koninklijke Bibliotheek, Den Haag, Nijmegen, 1995.

[41] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance Is Not Subtyping. *Principles of Programming Languages*, pages 125–135, 1990.

[42] T. Coquand and G. Huet. The Calculus of Constructions. Technical Report 530, INRIA, Centre de Rocquencourt, 1986.

[43] P. L. Curien and G. Ghelli. Coherence of Subsumption, Minimum Typing and Type Checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[44] H. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland Publishing Company, 1958.

[45] K. Czarnecki and S. Helson. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[46] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, 1978.

[47] R. Disetel. *Graph Theory*. Springer, 1997.

[48] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical Graph Transformation. *Journal of Computer and System Sciences*, 64:249–283, 2002.

[49] M. Dummett. *Elements of Intuitionism*. Oxford University Press, 1977.

[50] A. Egyed. Fixing inconsistencies in UML design models. In *Software Engineering, 2007. ICSE 2007. 29th International Conference*, pages 292–301. IEEE, 2007.

[51] A. Egyed, E. Letier, and A. Finkelstein. Generating and evaluating choices for fixing inconsistencies in UML design models. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference*, pages 99–108. IEEE, 2008.

[52] H. Ehrig, G. Engels, H. Kreowski, and G. Rozenburg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, 1999.

[53] P. Feiler, D. Gluch, and J. Hudak. The Architecture Analysis and Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, 2006.

[54] M. Fernández and J. Terrell. Assembling the Proofs of Ordered Model Transformations. In B. Buhnova, L. Happe, and J. Kofroň, editors, *Formal Engineering Approaches to Software Components and Architectures*, FESCA @ ETAPS, pages 63–78, Rome, Italy, March 2013.

[55] J. L. Fiadeiro. *Categories for software engineering*. Springer, 2005.

[56] K. Fisher, F. Honsell, and J. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[57] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[58] J. Girard. *Interprétation Fonctionnelle et Élimination des Coupres de l'Arithmétique d'Ordre Supérieur*. PhD thesis, Université Paris, 1972.

[59] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ 3. In *Conditional Term Rewriting Systems*. Springer, 1988.

[60] U. Golas, H. Ehrig, and F. Hermann. Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions. In R. Echahed, A. Habel, and M. Mosbah, editors, *Graph Computational Models*, CTIT Workshop Proceedings, pages 149–164, 2010.

[61] O. M. Group. MDA Guide Version 1.0.1, 2003.

[62] O. M. Group. Unified Modelling Language, Version 2.2, 2009. formal/09-02-02, formal/09-02-04.

[63] O. M. Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, 2011. formal/2011-01-01.

[64] L. Grunske, L. Geiger, and M. Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. In A. Hartman and D. Kreishe, editors, *Model Driven Architecture Foundations and Applications ECMDA-FA*, LNCS, pages 284–298. Springer, 2005.

[65] D. Halbert. *Programming by Example.* PhD thesis, Department of Electrical Engineering and Computer Sciences, Computer Science Division, University of California, Berkeley, 1984.

[66] Z. Hemel, L. Kats, and E. Visser. Code Generation by Model Transformation: A Case Study in Transformation Modularity. In A. Vallecillo, J. Gray, and A. Pierantonis, editors, *Theory and Practice of Model Transformations (ICMT 2008)*, volume 5063 of *LNCS*, pages 183–198. Springer, 2008.

[67] J. Hindley and P. Seldin. *Lambda-Calculus and Combinators.* Cambridge University Press, 2008.

[68] W. Howard. The Formulae-as-Types Notion of Construction. In J. Seldin and J. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press Inc, 1980.

[69] M. Jackson. *System Development.* Prentice Hall, 1983.

[70] I. Jacobson. *Object Oriented Software Engineering.* Addison-Wesley, 1992.

[71] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72:31–39, 2008.

[72] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA 2006, pages 719–720, 2006.

[73] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proceedings of the 2005 International Conference on Satellite Events*, MoDELS'05. Springer-Verlag, 2005.

[74] F. Jouault and I. Kurtev. On the interoperability of model-to-model transformation languages. *Science of Computer Programming*, 68:114–137, 2007.

[75] S. N. Kamin and U. S. Reddy. Two Semantic Models of Object-Oriented Languages. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming : Types, Semantics and Language Design*, chapter 4. The MIT Press, 1994.

[76] S. C. Kleene and J. B. Rosser. The inconsitency of certain formal logics. *Annals of mathematics*, 36:630–636, 1935.

[77] D. Kolovos, R. Paige, and F. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, GaMMa 2006, pages 13–20, 2006.

[78] I. Kurtev. State of the Art of QVT: A Model Transformation Language Standard. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, AGTIVE 2007, pages 377–393, 2008.

[79] K. Lano. Using B to Verify UML Transformations. In B. Baudry, D. Hearnden, N. Rapin, and J. Süß, editors, *Proceedings 3rd International Workshop MoDeV²a: Model Development, Validation and Verification*, 2006.

[80] K. Lano, D. Clark, and K. Androutsopoulos. RSDS: A Subset of UML with Precise Semantics. *L'Objet*, 9(4):53–73, 2003.

[81] H. Leiberman. *Watch What I Do : Programming by Demonstration.* The MIT Press, 1993.

[82] Z. Luo. *Computation and Reasoning.* Oxford Science Publications, 1994.

[83] P. Martin-Löf. An Intuitionstic Theory of Types: Predicate Part. In H. Rose and J. Shepherdson, editors, *Logic Colloquium.* North-Holland, Oxford, 1973.

[84] P. Martin-Löf. Constructive mathematics and computer programming. In L. Cohen, J. Pfeiffer, and K. Podewski, editors, *Proceedings of the Sixth International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, 1982.

[85] P. Martin-Löf. *Twenty-Five Years of Constructive Type Theory*, chapter An intuitionistic theory of types, pages 127–172. Clarendon Press, Oxford, 1995.

[86] J. Mesegeur. *A logical theory of concurrent objects*, volume 25. ACM, 1990.

[87] J. Mesegeur. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81:721–781, 2012.

[88] J. Meseguer. Membership algebra as a logical framework for equational specification. In *Recent Trends in Algebraic Development Techniques*, pages 18–61. Springer, 1998.

[89] M. Minas and H. J. Schneider. Graph Transformation by Computational Category Theory. In *Graph Transformations and Model-Driven Engineering*, pages 33–58. Springer, 2010.

[90] J. C. Mitchell. Toward a Typed Foundation for Method Specialization Inheritance. *Principles of Programming Languages*, pages 109–124, 1990.

[91] R. Monk. *Bertrand Russell – The Spirit of Solitude.* Vantage, 1997.

[92] J. Morris. Towards More Flexible Type Systems. In *Programming Symposium: Proceedings of Colloque sur la Programmation*, pages 377–384. Springer-Verlag, 1974.

[93] S. Muggleton and S. de Raedt. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 19-20:629–679, 1994.

[94] J. Myhill. Review of Foundations of Constructive Analysis by Erret Bishop. *Journal of Symbolic Logic*, 37(4):744–747, 1972.

[95] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[96] Object Management Group. Model Driven Architecture (MDA), 2001. ormsc/2001-07-01.

[97] Object Management Group. Meta Object Facility (MOF) Specification, 2002. 02-04-03.

[98] Object Management Group. Request for Proposal: MOF 2.0 Query / Views / Transformations, 2002. ad/2002-04-10.

[99] Object Management Group. Meta Object Facility (MOF) Core Specification, 2006. formal/06-01-01.

[100] Object Management Group. Unified Modeling Language (UML) Infrastructure, Version 2.4.1, 2011. formal/2011-08-05.

[101] Object Management Group. Unified Modeling Language (UML) Superstructure, Version 2.4.1, 2011. formal/2011-08-06.

[102] Object Management Group. Object Constraint Language (OCL), Version 2.3.1, 2012. formal/2012-01-01.

[103] A. Ohori and P. Buneman. Static Type Inference for Parametric Classes. In *Proceedings of ACM OOPSLA Conference*, volume 24, pages 445–456, 1989.

[104] P. C. Ölveczky, A. Boronat, and J. Meseguer. Formal semantics and analysis of behavioral AADL models in Real-Time Maude. In *Formal Techniques for Distributed Systems*, pages 47–62. Springer, 2010.

[105] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of real-time Maude. *Higher-order and symbolic computation*, 20:161–196, 2007.

[106] C. Paulin-Mohring. Inductive Definition in the System Coq - Rules and Properties. In M. Bezem and J. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, volume 664 of *LNCS*. Springer-Verlag, 1993. LIP research report 92-49.

[107] B. C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[108] B. C. Pierce. Bounded Quantification Is Undecidable. *Information and Computation*, 113, 1994.

[109] G. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 60(61):17–139, 2004.

[110] I. Poernomo. A Type Theoretic Framework for Formal Metamodelling. In R. H. Reussner, J. A. Stafford, and C. A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 262–298. Springer-Verlag, December 2006.

[111] I. Poernomo. The Meta-Object Facility Typed. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC 2006, pages 1845–1849, 2006.

[112] I. Poernomo. Proofs-as-model-transformations. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, volume 5063 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2008.

[113] I. Poernomo and J. Terrell. Correct-by-construction model transformations from partially ordered specifications. In J. Dong and H. Zhu, editors, *Formal Methods and Software Engineering*, volume 6447 of *Lecture Notes in Computer Science*. 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, Springer, 2010.

[114] T. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translators. *Computer and System Sciences*, 5:560–595, 1971.

[115] D. Rémy. Typing Record Concatenation for Free. *Nineteenth Annual Symposium on Principles of Programming Langauges*, pages 116–176, 1992.

[116] D. Rémy. Type Inference for Records in a Natural Extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming : Types, Semantics and Language Design*, chapter 3. The MIT Press, 1994.

[117] J. Reynolds. GEDANKEN – A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept. *Communication of the ACM*, 13:308–319, 1970.

[118] J. C. Reynolds. User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In D. Gries, editor, *Programming Methodology: A Collection of Papers by Members of IFIP WG2.3*, pages 309–317. Springer-Verlag, 1978.

[119] J. C. Reynolds. Using Category Theory to Design Implicit Conversions and Generic Operators. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, LNCS, pages 211–258. Springer-Verlag, 1980.

[120] A. Robinson. Review of Foundations of Constructive Analysis by Erret Bishop. *The American Mathematical Monthly*, 75(8):920–921, 1968.

[121] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modelling and Design*. Prentice Hall, 1991.

[122] B. Russell. *The Principles of Mathematics*. Cambridge University Press, 1903.

[123] B. Russell. Mathematical Logic as based on the Theory of Types. *American Journal of Mathematics*, 30(3):222–262, Jul. 1908.

[124] D. E. Rydeheard and R. M. Burstall. *Computational Category Theory*. Prentice Hall Englewood Cliffs, 1988.

[125] C. Schulz, M. Löwe, and H. König. A categorical framework for the transformation of object-oriented systems: Models and data. *Journal of Symbolic Computation*, 46:316–337, 2011.

[126] C. Schulz, M. Löwe, and H. König. Composition of Model Transformations: A Categorical Framework. In *Formal Methods: Foundations and Applications*, pages 163–178. Springer, 2012.

[127] A. Schurr. Specification of Graph Translators With Triple Graph Grammars. In E. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163. WG 1994, Spinger, 1994.

[128] S. Shlaer and S. Mellor. *Object-Oriented Systems Analysis – Modelling the World in Data*. Yourdon Press, Prentice Hall, 1988.

[129] S. Shlaer and S. Mellor. *Object Lifecycles – Modelling the World in States*. Yourdon Press, Prentice Hall, 1991.

[130] T. Skolem. Logico-combinatorial Investigataions in the Satisfiability and Provability of Mathematical Propositions. In J. van Heijnoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*. Harvard University Press, 1976.

[131] J. Terrell and B. Hawkins. *TA-5M Target Architecture*. Kennedy Carter Limited, `www.kc.com`, 2007.

[132] The Coq Development Team. *The Coq Proof Assistant, Reference Manual*, 2010.

[133] S. Thompson. *Type Theory & Functional Programming*. Computing Laboratory, University of Kent, 1999.

[134] D. Varró. Model Transformation by Example. In *Model Driven Engineering Languages and Systems (MODELS 2006)*, volume 4199 of *LNCS*, pages 410–424. Springer, 2006.

[135] D. Varró and Z. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):187–207, 2007.

[136] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodelling framework for describing mathematical domains and UML. *Journal of Software and Systems Modelling*, 2(3):187–210, 2003.

[137] E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Rewriting Techniques and Applications (RTA 2001)*, volume 2051 of *LNCS*, pages 357–361. Springer, 2001.

[138] M. Wand. Type Inference for Objects with Instance Variables and Inheritance. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming : Types, Semantics and Language Design*, chapter 4. The MIT Press, 1994.

[139] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 2003.

[140] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler. Towards Model Transformation By-Example. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, 2007.

[141] E. Yourdon and L. Constantine. *Structured Design*. Yourdon Press, 1978.

[142] M. Zloof. Query-by-Example: A Data Base Language. *IBM Systems Journal*, 16(4):324–343, 1977.

[143] S. Zschaler, I. Poernomo, and J. Terrell. Towards Using Constructive Type Theory for Verifiable Modular Transformations. In *FREECO 2011 Workshop Proceedings*, 2011.